O'REILLY
TURING 图灵程序设计丛书



前端架构设计

Frontend Architecture for Design Systems

让前端开发可持续优化、可扩展

[美] Micah Godbolt 著
潘泰燊 张鹏 许金泉 译
李弦 审校

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

译者介绍

潘泰燊

毕业于广东外语外贸大学，曾就职于腾讯、百度等互联网公司，参与过腾讯QQ空间、百度地图等海量数据业务的Web开发工作，现就职于富途网络。

张鹏

硕士毕业于中山大学，曾就职于百度、腾讯等知名互联网公司，参与过基于LNMP架构的亿级别互联网应用的设计与实现，目前从事NodeJS与前端开发。

许金泉

毕业于深圳大学，毕业后加入百度FEX，曾主导UEditor、百度国际化浏览器等前端开发工作，现就职于腾讯云。

审校介绍

李弦

华为2012实验室UCD交互设计师，Monash University交互设计硕士，前新东方英语教师，广东外语外贸大学英语专业八级。

TURING 图灵程序设计丛书

前端架构设计

Frontend Architecture for Design Systems

A Modern Blueprint for Scalable and Sustainable Websites

[美] Micah Godbolt 著
潘泰燊 张鹏 许金泉 译
李弦 审校

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（ＣＩＰ）数据

前端架构设计 / (美) 迈卡·高保特
(Micah Godbolt) 著 ; 潘泰燊, 张鹏, 许金泉译. -- 北
京 : 人民邮电出版社, 2017.5
（图灵程序设计丛书）
ISBN 978-7-115-45236-8

Ⅰ. ①前… Ⅱ. ①迈… ②潘… ③张… ④许… Ⅲ.
①计算机网络－程序设计 Ⅳ. ①TP393

中国版本图书馆CIP数据核字(2017)第060801号

内 容 提 要

本书展示了一名成熟的前端架构师对前端开发全面而深刻的理解。作者结合自己在 Red Hat 公司的项目实战经历，探讨了前端架构原则和前端架构的核心内容，包括工作流程、测试流程和文档记录，以及作为前端架构师所要承担的具体开发工作，包括 HTML、JavaScript 和 CSS 等。

本书适合前端开发人员，以及具有一定技术背景的前端管理者。

◆ 著　　[美] Micah Godbolt
译　　潘泰燊　张　鹏　许金泉
审　校　李　弦
责任编辑　朱　巍
执行编辑　温　雪　张　建
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：700×1000　1/16
印张：10.25
字数：243千字　　　　2017年 5 月第 1 版
印数：1－4 000册　　　　2017年 5 月北京第 1 次印刷

著作权合同登记号　图字：01-2016-9530号

定价：49.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版权声明

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第三部分　流程核心

第四部分　测试核心

第五部分　文档核心

前言

排版约定

本书使用了下列排版约定。

- 楷体

  表示新术语。

- 等宽字体（`constant width`）

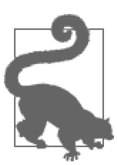
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`constant width bold`**）

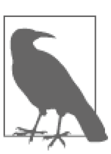
  表示应该由用户输入的命令或其他文本。

- 等宽斜体（*`constant width italic`*）

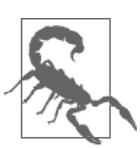
  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示。

本书适用对象

本书不是一本技术手册，虽然书中使用了大量的代码示例；它也不是一本纯理论图书，虽然讲“为什么这么做”的部分和讲“怎么做”的一样多。因此，本书既不适用于单纯寻求技术答案的开发者，也不适用于只想了解梗概信息的项目经理。

本书适用的对象是那些想从更宏观的角度理解前端开发的从业人员。我撰写此书的目的在于激励和鼓舞开发人员去承担起前端架构师的职责，以及力争在下一个项目中把前端开发作为头等重要的任务。

本书也同样写给那些具备一定技术头脑，并且想要理解当前日新月异的前端环境的管理者。本书涵盖了可以把项目前端开发水平提升到全新高度的多种工具、标准和最佳实践，并对此进行了有力的论证。

使用代码示例

补充材料（代码示例、练习等）可以从 https://github.com/micahgodbolt/front-end-architecture 下载。

本书的目的是要帮你完成工作。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如：“*Frontend Architecture for Design Systems* by Micah Godbolt (O'Reilly). Copyright 2016 Micah Godbolt, 978-1-491-92678-9.”

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。

用户可通过一个功能完备的数据库检索系统访问 O’Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网页地址是：

http://shop.oreilly.com/product/0636920040156.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：

http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

http://www.youtube.com/oreillymedia

电子书

扫描如下二维码，即可购买本书电子版。

第一部分

引言

Web 在诞生之初，形态其实是比较简单的。这里的“诞生”指的是 20 世纪 90 年代初期，而“简单”指的是一个网站能把自己的入口放在雅虎导航页上，并且访问者计数器能在使用表格布局并且满屏 GIF 动图的网页底部不停地闪烁。

不过，这的确是那个年代你所需要的一切。只要你能给 webring[1] 带来流量，所有的网站管理员都会满意。那些网站管理员是谁呢？没错，其实他们就是那些域名的拥有者。那时，网站还是简单的存在，管理员就是它们的主人。这些人习惯性地使用混乱的 HTML，还搞不清楚到底是不是应该关注新出现的层叠样式表（CSS），甚至他们当中大多数人已经把 JavaScript 作为过时的东西抛弃了。

不过，正如所有的媒介一样，网站也会进化。JavaScript 仍然被广泛使用，CSS 也不仅仅用于设置页面的字体类型和字体颜色。网站的管理员最终发现，他们走到了一个十字路口。网站的流量在持续增长，Web 技术也在持续发展（比如透明 GIF 等）。有太多的新事物和新技术需要学习，有太多的工作要做。最后，网站管理员不得不把网站的建设进行分工。一方面，他们很喜欢“网站正在建设中”这个标志以及到处可见的 marquee 标签；另一方面，Perl 是世界上最好的语言，毫无疑问它能够让网站永葆生机。

当网站管理员聘请另一个人来维护他们珍贵的域名时，他们需要决定自己是继续做一个写各种 `<blink>` 标签的大师，然后雇用写 Perl 脚本的新手，还是自己去学习 Perl。

注 1：webring，也称网络环，是相互连接的网站集合，通常主题是教育或社交，在 20 世纪 90 年代至 21 世纪初比较盛行，多数为非专业网站。——译者注

最终他们做出了决定，由此现代的 Web 开发团队开始成型，就像听到巨型海螺的召唤。早期的网站管理员在每一个阶段和十字路口，都会发现他们自己需要专注于开发过程中一个很小的部分。他们当中有些人会专注于服务器提供的文件服务，有些人想提高查询数据库的能力，而另外一些人则乐于创作各种图形和图像。

更新的、更专业的角色也吸引了其他人到这个行业，包括艺术家、作家、商业分析师、工程师、数学家等。随着这些角色的发展，以及相关的人员越来越成熟，Web 开发逐渐催生了一系列新的名称和新的分支学科。

那些决策者

在 Web 开发的早期，有些人认为网页中的文本内容跟设计、代码甚至搜索引擎优化（众所周知，搜索引擎就是抓取页面中的关键字）一样重要。在此之前，网页内容往往是被放到后面处理的。“先随便填一些 lorem ipsum[2] 字符到设计稿里，赶紧做后面的。”最终在网站上线之前，客户才会填上真实、优质、受启发的内容，并且一直以来都是如此。

这些内容的拥护者，终于坚定直接地宣称 Web 的本质就是内容，因此网站内容值得我们花费时间和精力。尽管这是一场恶战，不过他们开始被邀请参加早期的规划会议，偶尔也被邀请参与编辑策略的制定。他们不断取得进展！虽然这个过程困难而且孤独，但是结果是值得的。

就这样，他们作为开拓者在孤独地前进。直到关键的一天，他们碰巧遇到另外一个同样拥护内容的人，才意识到原来他们不是在孤军作战。友谊的星火以燎原之势发展成方方面面的合作，最终他们成立了社区，继续致力于向人们宣传网站内容的重要性。

很多年过去了，但是关于网站内容的争论远远没有结束。即使多一个设计师被要求在主页上“随便填充点内容”，我们也能马上听到远处抗议的呐喊声。2008 年 12 月 16 日，Kristina Halvorson 在 A List Apart 博客（http://alistapart.com/article/thedisciplineofcontentstrategy）上站出来发声，举起了内容策略的大旗。她要求我们传播这种理念，“要把网站内容作为一个值得做出战略规划和有投资价值的关键元素来对待”。其他内容策略的践行者们开始学习、使用和推广它，成为了内容策略师。由此，一个新的分支学科诞生了。

Kristina 的文章并不是最早提出内容策略这个概念的，却是最早定义出内容策略的核心、精神和目标的。一夜之间，内容拥护者们的共同诉求被赋予了一个名字。他们即将迎来一个新的时代，博客、播客和会议都围绕着一个简单的观点来讨论——“内容很重要”。

注 2：lorem ipsum，中文又称“乱数假文”，是指一篇常用于排版设计领域的拉丁文文章，主要的目的为测试文章或文字在不同字型、版型下的效果。——译者注

响应式Web的出现

与此同时，一个身着黑色高领毛衣的男人出现了，并且颠覆了人们对互联网接入设备的理解。在 Web 历史上，我们第一次被迫接受这么一个事实：人们并不是只有坐在舒适的办公室或起居室，使用高速的宽带网络，对着分辨率为 1024 像素 ×768 像素的电脑屏幕时，才会去浏览网站。iPhone 的出现，使人们迎来了一个多分辨率、多功能、连接速度不稳定以及多种输入模式的多终端时代。作为开发者，我们不能再对用户和他们用来浏览网站的设备做出任何假设。

为此，我们尝试过很多解决方法。我们尝试使用双指缩放、双击缩放、直接显示原网站，或者把移动设备重定向到一个精简的便于移动浏览的"m.dot"网站。然而，没有一个方案能够真正解决问题。双指缩放的操作不便于导航，尤其对于购买商品或注册服务，而且增加移动流量意味着损失收益。虽然 m. 网站对移动设备更加友好，但是这样就要求开发团队去维护两个独立的网站。

很多 m. 网站被冷落了，它们也很难和 PC 版主站保持同步更新，或者部分功能缺失而迫使用户切换到桌面设备，来做一些比获得指引或者打个电话更复杂的事情。我们必须做出一些改变。虽然有些人觉得 iPhone 已经过时，但是很明显，Web 世界的未来在移动设备上。

2010 年 5 月 25 日，iPhone 发布 3 年后，Ethan Marcotte 在 A List Apart 上发表了一篇题为"Responsive Web Design"的长文（http://alistapart.com/article/responsive-web-design）。这篇文章并没有介绍新的学科知识，也没有打出口号，让困境中的开发者聚集起来；而是描述了一种创建新型网站的方式，这些网站可以根据用户访问设备的尺寸来做出响应，并且调节自身大小来适配对应的视口。响应式 Web 设计（responsive Web design，RWD）不是一种新科技，而是现有工具和技术的一个集合，包括以下内容。

流式网格

用基于百分比的宽度代替固定像素的尺寸。

自适应图片

用 100% 宽度的图片填充它们的容器，并且随着视口大小的改变而改变。

媒体查询

允许对不同的视口大小使用不同的样式，我们现在可以基于屏幕的大小来改变页面布局。

在 Ethan 发表这篇文章的几年之前，所有的这些技术其实早就可以在浏览器中使用。不过，正如 Kristina 为内容策略发起的号召，Ethan 对 RWD 的描述清晰地定义了每个人都在迫切寻找的解决方案。

一篇简单的文章，使 Web 开发这个行业发生了变化。

前端架构的种子

根据这段历史，我开始思考什么是前端架构。作为一个 Drupal 的前端开发人员 [3]，我对那些内容战略家过去所面临的窘境感同身受。编写前端样式总是作为延后的事情来考虑，它是在设计师和后端开发人员完成工作之后，再给默认的标记加上一层漂亮的外表。对于我们所面临的挑战，没有什么比不同角色进入项目的先后顺序更能说明问题。我见过一些项目在启动之后，先是讨论设计方案，然后开发功能，最后才把前端开发人员加入到项目中，让他们把设计师扔过来的各种设计实现成 CMS 输出的标记。

经历了几次这样的流程之后，我知道当我尝试去解构那一堆移动端和桌面端的设计稿时，我将会非常痛苦。这样做的目标是把它们制作成一个主题，应用到 Drupal 输出的一堆 div 标签上。跟一些使用 Rails 的朋友谈及编写网站导航样式所面临的挑战时，我迫不及待地承认，“开发者不愿意对导航的标记进行哪怕是一点点的修改”，而事实的确如此！一旦这些标记被定下来，开发人员马上就会进入到下一个任务，这时候如果想修改一下 div、列表或者链接，那简直是痴心妄想。毫无疑问，这将会产生一些奇怪的 CSS hack 去实现某些设计，而这些设计往往都难以跟实际生产环境中的默认导航标记相匹配。

多年以来，前端开发人员的价值体现在创建弗兰肯斯坦风格的设计模式的能力上。“现在，如果我给第三层嵌套的 div 添加一个伪元素，并且给它设置一个雪碧图的背景图……”，这是我们的技术方案，它很糟糕。我们一直在填坑，还奢望能够赶在技术债务把我们拖垮之前把网站发布出去。

随着项目复杂性的增长，这样的流程不是长久之计。所以，我开始思考，如果不再使用传统的方式去开发（因为以前这种方法一直都是可行的），而是把前端开发当作“一个值得做出战略规划和有投资价值的关键元素”，会使项目产生什么样的变化。如果我们在 CSS 框架、文档工具、构建流程的命名规范，甚至标记本身这些方面拥有话语权会怎样？我开始思索如果是 UX 开发去主导后端开发，而不是反过来，那么一个大规模项目会变成什么样子。

这样能引起一场变革吗？其他人愿意跟随并开始“学习、使用和推广它”吗？在形成一股力量之前，我们需要了解目标。我们争取的是什么？我们怎样实现目标？我们会被赋予什么样的称呼？

前端架构师的含义

在后端开发领域，系统规划和可扩展性非常关键，因此软件架构师备受重视。早在开发工作启动之前，他们就被邀请加入到项目中，而且他们会跟客户讨论即将建成的平台的架构

注 3：Drupal 是一个由 Dries Buytaert 创立的自由开源的内容管理系统，用 PHP 语言写成。——译者注

要求。他们将会使用什么技术栈？内容类型是什么？这些内容如何被创建、保存以及展示在屏幕上？软件架构师的职责就是要保证项目中每一步都在总体架构的指导下进行，而不会随机决定。

我意识到前端开发领域缺少的就是架构。我们总是被要求做一些零碎的工作，而且优先级也不高。我没用多长时间就从数据库和 Web 服务器切换到 Sass 文件夹结构和构建系统，因此前端架构师的头衔就这样诞生了。

现在，任何一个职位名称都需要一个职位描述。前端架构师是做什么的呢？如果有合适的机会，他们会对项目产生什么样的影响？这些思考促使我在公司年会上做了一场关于前端架构的简短演讲。另外，在 CSS 开发者大会上的发言机会也让我将想法认真总结成了一个精简的 45 分钟演示。

2014 年 10 月 13 日，在新奥尔良会议中心一个拥挤的房间里，“举起前端架构的旗帜”成了在一线奋斗着的开发者们共同的诉求。我希望他们知道，他们不是孤军作战，有很多人在支持他们。我跟项目经理、销售人员和开发者沟通，给他们描绘拥有良好前端架构的意义，以及它为团队和客户带来的价值。

在这次演讲之后，我听到了很多故事，都是关于那些终于弄清楚自身定位以及在公司中所扮演的角色的开发者们的。很多人现在才发现，其实他们一直扮演着前端架构师的角色，却从来没有拥有过这个头衔，或者没有足够的信心去争取这个职位所应具有的权力。在 CSS 开发者大会召开几周之后，我发现很多人把他们在 Twitter 上的个人简介改成了“前端架构师”。而我，作为他们中的一员，从那以后，就在这条路上坚定地走下去了。不管我现在的工作头衔是什么，我都是一名前端架构师。

第1章 前端架构原则

> 前端架构是一系列工具和流程的集合，旨在提升前端代码的质量，并实现高效、可持续的工作流。

当思考前端架构师的角色时，我总会联想到传统的建筑设计师。

在建设过程中，建筑设计师需要设计和规划方案，并且跟进施工过程。这与前端架构师的工作有着异曲同工之妙，不同的是后者建造的是网站，而不是建筑物。比起浇筑混凝土，建筑设计师会在设计工程构图的工作上倾注更多的精力。同理，相比编写具体的代码，前端架构师更专注于开发工具和优化流程。

下面，我们来深入分析前端架构师的工作职责。

体系设计

试想一下，如果一栋建筑没有明确的构造设计，所有的重要事项都由建筑工人直接决定，那么就可能会出现这样的情景：第一面墙用石头垒，第二面墙用砖头砌，第三面墙用木头搭，第四面墙因为追求时髦而留空。

虽然网站的整体外观和风格基调完全由经验丰富的视觉设计师决定，但前端架构师掌控着背后的前端开发方法和系统设计哲学。通过设计所有前端开发人员都要遵循的系统规范，前端架构师清晰描绘了产品和代码的最终形态。

一旦前端架构师建立起了系统设计的规范，项目就拥有了可以衡量代码质量的标准，否则我们如何判断代码是否达标呢？一个精心设计的系统，应当具备完善的检验机制，并做出适当的取舍，以保证系统中的代码有实质的价值，而不是简单的堆砌。

工作规划

有了清晰的结构设计之后，就需要制定开发工作流了。开发人员写一行代码并且提交到线上需要经过什么步骤？举一个最简单的例子，这个过程包括使用 FTP 登录服务器，修改一个文件并保存。然而，对于大多数项目而言，完整的工作流可能会用到多种工具，如版本控制器、任务调度器、CSS 处理器、文档工具、测试组件和服务器自动化工具等。

前端架构师的目标是设计出能流畅运转的系统。这个系统不仅能高效快速地启动，还可以通过语言分析、测试用例、文档记录等方法持续地提供有效的反馈，并且大幅减少由于重复操作而产生的人为错误。

监督跟进

前端架构设计绝不是一劳永逸的工作。没有任何设计在一开始就是完美的，也没有任何计划可以一步到位。客户和开发人员的需求会随着时间改变。在某个阶段运行得很好的开发流程，随后也可能需要重新调整，以便提高效率、减少错误。

前端架构师的一个非常重要的能力，就是能够持续地优化工作流程。如今各种各样的构建工具可以让我们很方便地改变工作方式，并通知到每一位开发人员。

有些人问前端架构师是否等同于管理角色，不再需要写业务代码。我以过来人的身份向你保证，前端架构师不仅要写更多代码，更要会用多种编程语言，还要使用大量的工具。代码量并未减少，只是代码的读者发生了改变。前端开发人员面向终端用户写代码，而前端架构师面向的则是团队里的开发人员。

像盖房子一样搭建站点

前端架构需要争夺属于自己的优先权，正如我们很难想象有人会在不咨询建筑师的前提下，就建造一座摩天大厦。但事实上，很多大型的网站项目就是这么直接开始的。

借口数不胜数："我们预算不够了""哪有时间做这个""等设计做完了再说吧"；甚至更糟糕的是，你只是被毫无理由地突然空降到项目组——所有的设计已经定稿，开发工作也如火如荼地进行了几个月，而你只有几个月的时间把别人扔过来的一堆设计稿奇迹般地且完美地实现成一个个的网页。这当然不可能开发出实用性强、可扩展且稳定强大的网站。

作为前端架构师，我们认为有多个关键的决策需要在项目启动之初就制定下来。如果等到开发阶段的后期再考虑，不是已经用不上了，就是一开始错误的决定已经造成了无法挽回的损失。一旦做出这些决策，我们的任务就是去辅助视觉设计、平台开发、底层结构，使之能最大程度地满足需求。

如果没有前端架构师的提前介入，项目就有可能陷入两难境地：或是将视觉设计、平台或

底层结构推翻重做，或是让前端开发人员自己去克服困难。经验告诉我，推翻重做通常不会有什么好结果。

有什么难题

我知道，这不是一个轻松的任务。我所提议的改变需要耗费不少的成本，而且任何一个负责做这些决定的人都需要权衡各种利弊。对于那些没有与前端架构师一起工作过的人来说，这将会存在很大的风险。

正如人们争论“先有鸡还是先有蛋”的问题，我们面临的窘境是，如果要说服老板们为一个完善的前端架构系统花费时间和金钱，他们往往会要求提供这种架构的成功案例，而这显然需要你曾经参与过类似的项目。问题是，如果你总是被要求证明这种工作流程是有效的，又怎么会有机会去实际地为某个项目设计前端架构呢？

对于我和这本书来说，很幸运，我最近被委以重任，为一个大型的网站建立新的架构系统。我有充足的时间思考和规划这个新系统，并制定新的代码标准、工具和开发流程。一旦这个项目开始成形，我就得到了一个很可贵的机会，去证明合适的架构设计有多么强大的可扩展性和可持续性。

第2章

Alpha项目

在过去的一年里，我有幸以架构师的身份参与了 Red Hat 公司的一个项目，该项目希望将已有的网站组件共享到公司的多个网站中。在构建 Redhat.com 一段时间之后，我知道自己还是面临很多挑战。我花了一些时间建立起开发团队对我的信任，最终得以全权负责整个项目的架构。

这个机会非常难得，因为有了它，我们才能解决“先有鸡还是先有蛋”的问题。我的团队有机会去搭建一个庞大的、有足够技术支撑的设计方案，并且整个方案最终会被展现在一个流量很大的站点中。这将会成为一个标志性的项目，能够让我在后续其他项目中推广前端架构设计的理念。

2.1 慢而有力的开端

能够在这个时候与 Red Hat 公司合作，我的团队实在是太幸运了。我们对旧站点进行了重新设计之后，并没有马上着手开发新功能。虽然我们时不时也需要处理一下 bug，但毕竟还是有几个月的时间去研究如何为这个系统设计全新的架构。

尽管在 Redhat.com 生产环境中仍然有大量的遗留代码，我们还是选择了从零开始。因为原来的站点是基于块（一行接一行的独立内容）设计的，所以我们新开发的网页元素可以布局在已有内容块的上方或下方。我们不需要重写侧边栏的样式，也不需要频繁地改变 HTML 标签的样式。实际上，我们相当于在旧站点中构建全新的网站。正如船上的甲板可以一块一块地被替代，我们希望最终也可以替换原有的系统，完全使用新的系统。

既然自由发挥的空间这么大，我们完全可以创建一张很长的愿望清单，并且真的去实现它

们。这个愿望清单包括以下方面。

模块化内容

　　我们非常推崇 Brad Frost 提出的原子设计方法论（http://patternlab.io），希望尽可能复用小的组件，而不是弄出几十个、甚至上百个不同的内容块。

全面测试

　　我们之前经常出现这样的情况：大量的前端代码合入到主干，然后导致几个月前的代码出现运行问题。这样太浪费时间了，所以我们决定要像测试后端代码一样测试我们的新框架，达到一样的代码覆盖水平。

流式处理

　　我们希望引入 Git 工作流程，用它来管理应用代码简直是游刃有余，但是我们要将功能分支分解为更小的、模块化的代码块。另外，也要把过去一直采用的容易出错的手动步骤自动化：更新样式表、创建图标字体、部署新代码等。

详细的文档

　　这个新系统的受众很广，包括前端开发人员、后端开发人员、设计师、市场经理、运维人员，以及其他产品开发角色。我们希望每个人接触这套系统时，都能找到适合自己的、详细的文档。

这四个方面需要一点时间去开发、配置和完善。在最初几个月里，我们的开发效率确实不如以前。但是一旦新系统有了一定的基础，我们很快就感受到了它的优势。

当再次遇到包含 2 个，甚至多达 12 个内容块的页面需求时，我们发现工作量比以前少得多。我们不再需要独立对待每一个内容块，而是将内容块拆分为最小的独立可重用组件，并根据需求决定是否新增布局或组件。

最终，每一个需求都会产出详尽的文档、全面的回归测试方案以及符合最初架构设计规范的源代码。

虽然这些工具和流程需要一些时间来打造，但是效果实在太好了，现在我们已经开始嘲笑自己的工作有多么简单了。当然，工作简单是好事，这意味着我们可以少做一些重复的工作，把更多时间花在系统设计上。最终，在我们设计的系统中工作会感到愉悦，面对每一个用户需求，我们都会感到兴奋，因为又有机会将系统变得更强大了。

2.2　全副武装

有了这些经验后，我相信我们编写的代码、开发的流程、磨练的技术能够充分验证这套方法论的合理性，并成为说服其他人在项目中使用它的坚实基础。

使用本书提到的流程、技术和书中讨论的经验，我希望你在下一个项目中能够有自信为自己的前端架构而战。争取尽早参与项目，以能够在项目的重要决策中发挥影响力；挑战现有的工具和流程，构建更智能、可重用性更高的代码；争取使你的前端工作产生更大的影响力。扛起前端架构师的旗帜，和我一起战斗吧！

第 3 章

前端架构的核心

任何一栋建筑都需要稳固的基础、四面墙体和一个屋顶。这些要素都是必不可少的。基础支撑着墙体，墙体支撑着屋顶，而屋顶保证你安全并且免受风吹雨淋之苦。如果一名建筑师不能提供上述要素，那么他必定是不称职的。作为前端架构师，我们在构建新网站时也承担着相似的责任。我们必须驾驭必要的工具和流程，而这两者正是成功构建网站的要素。

3.1　围绕四个核心工作

本书接下来将讨论前端架构的四个核心。这四个核心的主题、技术和实践是构建可扩展和可持续优化的系统的基础。它们引发了在任何前端开发项目中都需要进行的一系列讨论。这些讨论会帮助我们确立对项目的整体期望，包括代码质量、实现每一项需求所需的时间和工作量，以及保证所有开发工作能够按时完成的工作流。

当然，本书所探讨的四个核心绝不是唯一的方法，甚至都算不上最好的方法。每一个决定都应该视项目的实际情况而定，有时候最好的决定甚至是什么都不做。不同于世界 500 强公司的那些需要持续多年面向客户的项目，小规模或者临时过渡的项目并不需要特别复杂的基础架构。

不要以为掌握了接下来的内容就可以高枕无忧。前端架构师的成长之路不是一蹴而就的，而是需要保持不间断的学习状态。这种状态决定了我们的水平和价值。对于前端开发领域的广泛涉猎使我们能够很快上手各种新技术和方法论。

我们的强项之一是花一个小时就能了解某个新框架或者 Gulp 插件，找出其亮点和不足，并确定它在项目中的可行性。因此，如果你迷失在本书接下来的大量技术和概念中，请记

住：并没有人精通每一种技术。就我个人而言，我也只精通这些内容的一小部分，胜任一大部分，而余下部分则是入门水平。

好了，关于这些核心的介绍差不多了。接下来让我们深入分析它们的具体含义。

3.2 四个核心的含义

代码

归根结底，所有的网站都是由一堆文本文件和资源文件组成的。当我们面对制作网站所产生的大量代码时，就会发现为代码和资源设定一个期望是多么重要。

在代码部分，我们会专注于如何实现系统架构中的 HTML、CSS 和 JavaScript。

流程

既然早已过了 FTP 上传文件的时代，那么现在重要的是思考怎么用工具和流程构建一个高效且避免出错的工作流。工作流变得越来越复杂，那些用于构建它们的工具也同样如此。这些工具在提高生产力、加快效率和保持代码一致性上带来了惊人的效果，但也伴随着过度工程化和抽象化的风险。

正如工作流在演变，工作的方式也在进步。我们不再浪费时间把一些 Photoshop 设计稿重构成 CMS 模板页面。因为逐渐把设计的环节转移到浏览器中，并书写响应式的网页框架，所以在实现 CMS 的界面之前，我们往往已经开始编写所有的 HTML 和 CSS 代码。要实现这个颠覆性的角色转变，就需要改变现有的开发流程。

测试

要构建一个可扩展和可持续优化的系统，必须保证新代码与老代码能够很好地兼容。我们的代码不会独立存在，它们都是大型系统中的一部分。创建覆盖面广泛的测试方案，能确保老代码还能正常运作。

在这个部分，我们会探讨三种用于测试网站的不同方法。根据团队的规模，测试方案有时会细分为很多部分，如前端、后端和运维。而深入理解每个部分将会有助于你和其他团队高效沟通。

文档

一般而言，如果不是团队中的重要成员要离开，我们几乎都不会意识到文档的重要性。等到那个时候，大家将不得不停下手头的工作，优先编写所有的文档。作为前端架构师，你要善于在项目开发的同时编写良好的文档。

这个部分将会介绍团队需要编写的各种文档类型、发布文档的工具，以及阅读这些文档的用户角色。

第二部分

代码核心

好程序绝非偶然天成。这并不是说开发人员天生懒惰或者不值得信赖，而是因为独立工作的时候，我们针对同一个问题能提出各种不同的解决方案。不同于走迷宫，解决问题几乎不会只有一种方法。我们每个人的经验、观点和习惯各异，因此解决同一个问题的方式也不尽相同。

开发人员意见纷纭是很正常的。这种从不同角度看待问题的能力使得我们团队越来越强大。但是在输出方案并且实际应用于设计系统的时候，我们既不希望也不需要在程序中反映出这些差异。

即使开发人员都能以同样的方式解决问题，我们也不能保证代码适用于系统的其他部分。这就无异于开发人员写出一个精致优雅的 Bootstrap 主题，但项目需要的却是个性化主题。

这一部分将帮助我们探讨如何提高 HTML、CSS 和 JavaScript 的代码质量，编写类、设计函数，以及声明接口。

接下来的章节并非面面俱到。作为一名前端架构师，你的工作是不断地探索和评估新的技术、平台、方法和框架。世界上没有一刀切式的解决方案，而前端架构师的使命正是将项目的需求与前端开发的实际情况相结合。

第 4 章 HTML

作为前端架构师，你面临的第一个挑战就是标记的规范化，换言之，你希望开发人员写出或 CMS 产出什么样的标记。如果没有 HTML，Web 就不复存在。因此我们暂且把 CSS 和 JavaScript 放一边，先单独看原始的内容和控件。

文字、图片、链接、表单和提交按钮都是所有 Web 真正需要的元素，也是在 Web 上创建所有东西的基础。初始的标记做得不好，你将要写很多不必要的 CSS 和 JavaScript 来弥补。而初始的标记做得好，你就能写出更具可扩展性和可维护性的 CSS 和 JavaScript。

4.1　过去处理标记的方法

过去，不少前端从业人员来自出版界，或者曾与出版界的设计师、产品负责人共事，因此 HTML 的布局跟手册、杂志或报纸等传统材料非常相似。现如今二者不能等同了。但是在响应式设计成为业界标准之前（甚至之后的一段时间），大多数网络媒体资源都被视为多页印刷品的集合。项目分工之后，你会被指定负责某个页面，从头开始创建 DOM。

过去，我们的标记通常被细分为两大阵营：程序式和静态式。下面就来看一看。

4.1.1　程序式标记：自动化程度100%，可控程度0%

在 Web 出版领域，前端团队没有标记的控制权是非常普遍的。这通常是由于在前端团队参与项目之前，项目的功能开发（包括 HTML 输出）就已经进行了几周甚至几个月了。如果标记源码被复杂的渲染过程打乱，而且还来自不同的模板，那么情况就会变得更加糟糕。

这意味着，对于任何不熟悉 CMS 后端复杂性的人而言，更新标记将会异常困难。而往往到了这个时候，后端开发人员已经开始着手别的任务了，没有时间回过头来进行任何重大的修改。

这种制约因素的影响是，为了把内容更好地嵌入到 HTML 中，CMS 编辑人员和后端开发人员宁可写一堆标记和 CSS 类名。最终，他们编写的代码如下：

```
<div id="header" class="clearfix">
  <div id="header-screen" class="clearfix">
    <div id="header-inner" class="container-12 clearfix">
      <div id="nav-header" role="navigation">
        <div class="region region-navigation">
          <div class="block block-system block-menu">
            <div class="block-inner">
              <div class="content">
                <ul class="menu">
                  <li class="first leaf">
                    <a href="/start">Get Started</a>
```

Drupal.org 首页的这一段代码显示了一个简单的页面顶部，即使在填充内容之前都可以包含 10 层嵌套，更可怕的是，在实际应用中这还不是一个特别夸张的现象。经验告诉我们，它还可能嵌套得更多。

以前，这种“div 乱炖”或许确实有助于我们把静态 Photoshop 图像做成标记化的页面，但随着我们的需求日渐成熟，我们急需更好的方法来驾驭它们。

4.1.2　静态标记：自动化程度0%，可控程度100%

如果我们的项目规模比较小，或者任务只是开发一个需要填充一大块主体区域的页面，那么编写静态标记更为方便。虽然这种情况灵活性很大，但是也意味着我们必须负责维护所有的代码。一个原本在 CMS 模板中很容易的改动，现在需要每个页面单独手动修改。所以我们会写成这样：

```
<header>
  <section>
    <nav>
      <div>
        <ul>
          <li>
            <a href="/products">Products</a>
            <ul>
              <li>
                <a href="/socks">Socks</a>
```

为了保持简洁，“语义化”的标记是首选，应用样式所依靠的是 HTML5 元素名称和它们的层级关系，而非 CSS 类名。我们的标记中没有 CSS 类名，主导航的样式会自动继承到二级导航的锚点上，我们深受其害，最终往往写出这样的后代选择器：

```
header > section > nav > div > ul > li > a {
  color: white;
}
header > section > nav > div > ul > li > ul > li > a  {
  color: blue;
}
```

过去，这种静态标记的方式使得对于任何一个悬停状态或激活状态选择器，代码至少得写这么长。你根本不想看三级导航会被写成什么样。我们还是赶紧跳过这部分，看一下现在是如何处理的。

4.2　平衡可控性和自动化

作为前端架构师，你需要评估标记产生的过程。你对内容的顺序、使用的元素和 CSS 类名有多大的控制权？这些元素在将来改动起来会有多大难度？模板是否易用，或者是否只有后端开发人员才能更改？甚至，你的标记全是基于模板系统的吗？你可以通过系统做出更改，还是需要手动处理？通过回答这些问题，你可能会颠覆自己构建 HTML 和 CSS 的方法。

模块化标记：自动化程度100%，可控程度100%

我们都在追求的理想状态是，网站上每一行 HTML 都由程序自动生成，而作为前端开发人员，我们只需要管理这个用来产生标记的模板和流程。遗憾的是，现实通常并非如此。即使在最好的情况下，也存在用户生成的内容，而这些内容几乎都无法自动添加 CSS 类名来标记。无论 CMS 系统自动生成 HTML 的能力如何，让 CMS 决定类似表单和导航栏这样的标记，有时候会更简单。但是就算你已经把理想状态实现了 90%，标记的模块化方案仍然可以给你带来理想的灵活性和必要的自动化。

模块化标记和程序化标记的区别在于，对于使用什么标记输出既定内容，我们不会完全任由 CMS 决定。这使得我们可以为两个不同的导航实例使用一样的标记，虽然 CMS 生成的标记可能完全不一样。模块化标记和静态化标记的区别在于，程序化地执行完之后，我们还可以通过一套类名系统给标记动态添加 CSS 类名，并且不再通过元素标签和层级关系来决定视觉外观。让我们看一下如何用 BEM 原则模块化地实现一个简单的导航：

```
<nav class="nav">
  <ul class="nav__container">
    <li class="nav__item">
      <a href="/products" class="nav__link">
        <ul class="nav__container--secondary">
          <li class="nav__item--secondary">
            <a href="/socks" class="nav__link--secondary">
```

乍看上去，这种方案似乎相当冗长。这一点我没有什么好辩解的，但我要说的是，它的冗余程度其实是恰到好处的。给每个元素都添加了相应的 CSS 类名之后，我们就不再需要依

赖那些只为了样式标签而存在的 CSS 类名或元素的层级关系来决定视觉外观了。相比动态标记，这个标记更清晰，并且我敢说，这也让标记的组织形式更“模块化”了。这个导航可以作为网站的导航通用模板，不用改任何一个标记就可以在多处复用。因此，这种标记并不是先等 CMS 创建完成再另外添加样式标记的，而是创建的同时就添加了样式标记，然后整合到网站的整个导航系统中。

4.3 这一切背后的设计系统

要使用这种模块化方法，我们首先需要改变构建页面的方法和思路。单独的静态“网页”其实根本就不存在。所谓的“网页”其实是过去的产物。那么什么是当前我们真正需要去创建的“页面”呢？是某个 URL 的内容吗？如何保证每次访问某个 URL 的时候都访问到同样的内容呢？如果你登录了会如何呢？如何使这个 URL 的内容根据时间、地点或者浏览器的不同而有所不同呢？越早地意识到我们现在所做的工作不是单纯地实现某个页面，而是设计整个系统，就能越早地开始创造那些真正让人惊艳的 Web 作品。

设计系统是网站视觉语言的程序化呈现。设计师创造的视觉语言，是用来描述网站如何通过视觉与用户沟通的工具。它集合了颜色、字体、按钮、图片样式、排版布局和界面版式，用来传达情绪、意义和目的。

正如词可以分为名词、动词和形容词等，作为前端开发人员，我们的工作就是把视觉语言拆解成最小单元。拆解之后，我们可以创建规则，对这些最小单元进行重组。只有将视觉语言充分拆解之后，我们才能知道如何重新把它们连成“句子”，组成“段落”，合并为“章节”，最后创作成一部精彩的“小说”。转换的目标是创建具有可扩展性和可维护性的代码库，以便如实地重现视觉语言能表达的任何东西。

在后面的章节中，我们将进一步创建设计系统，但是在目前的阶段，认清我们正开发的是什么很重要，因为在着手开发之前，我们需要先决定设计系统要如何建立在标记之上。

4.4 模块化CSS理论的多面性

如今，CSS 理论几乎和 CSS 或 JavaScript 框架一样多。但 CSS 或 JavaScript 框架的用法较为繁琐，而且必须成套使用，而 CSS 理论更多的是阐释 HTML 和 CSS 之间的关系，而不是预编译的代码库，因此使用起来更为灵活。

你好像每天都会听说一个新的方法，例如使用新的命名空间、扩充数据属性，甚至是在 JavaScript 里定义 CSS。这些理论都有它的亮点，能够在如 HTML 和 CSS 的关系方面给你一些新的启发。

当然，没有哪个方法论是完美的，你可能会发现，一个项目与某一个方法契合得最好，但

是另一个项目却更适合用另一个方法。所以，你完全可以创造自己的方法论，或者将一个现有的理论根据自己的需求进行改造。因此，如果你犹豫不决，不知道如何选择方法论，最好是看一些比较杰出的方法论，根据你手头的项目来分析其中哪些可用，哪些不可用。

4.4.1 OOCSS方法

下面的代码片段展示了如何使用 OOCSS（Object-Oriented CSS，面向对象的 CSS）方法创建一个 HTML 切换。

```
<div class="toggle simple">
  <div class="toggle-control open">
    <h1 class="toggle-title">Title 1</h1>
  </div>
  <div class="toggle-details open"> ... </div>
  ...
</div>
```

OOCSS（http://oocss.org/）有两个主要的原则：分离结构和外观，以及分离容器和内容。

分离结构和外观，意味着将视觉特性定义为可复用的单元。前面那段简单的切换就是一个简短的可复用性强的例子，可以套用很多不同的外观样式。例如，当前的“simple”皮肤使用直角，而“complex”皮肤可能使用圆角，还加了阴影。

分离容器和内容，指的是不再将元素位置作为样式的限定词。和在容器内标记的 CSS 类名不同，我们现在使用的是可复用的 CSS 类名，如 `toggle-title`，它应用于相应的文本处理上，而不管这个文本的元素是什么。这种方式下，如果没有应用别的 CSS 类名，你可以让 H1 标签以默认的样式呈现。

当你想提供一套组件让开发人员组合起来创建用户界面时，这种方法非常有用。Bootstrap 就是一个特别好的例子，它是一个自带各种皮肤的小组件系统。Bootstrap 的目标是创建一个完整的系统，而且这个系统能够创建开发者可能需要的任何用户界面。

4.4.2 SMACSS方法

同样以切换组件为例，按照 SMACSS（Scalable and Modular Architecture for CSS，模块化架构的可扩展 CSS）方法，写出来的代码如下：

```
<div class="toggle toggle-simple">
  <div class="toggle-control is-active">
    <h2 class="toggle-title">Title 1</h2>
  </div>

  <div class="toggle-details is-active">
    ...
  </div>
  ...
</dl>
```

尽管 SMACSS（https://smacss.com/）和 OOCSS 有许多相似之处，但 SMACSS 的不同点是把样式系统划分为五个具体类别。

基础

　如果不添加 CSS 类名，标记会以什么外观呈现。

布局

　把页面分成一些区域。

模块

　设计中的模块化、可复用的单元。

状态

　描述在特定的状态或情况下，模块或布局的显示方式。

主题

　一个可选的视觉外观层，可以让你更换不同主题。

在前面的例子中，我们看到了模块样式（`toggle`、`toggle-title`、`toggle-details`）、子模块（`toggle-simple`）和状态（`is-active`）的组合。对于如何创建功能的小模块，OOCSS 和 SMACSS 有许多相似之处。它们都把样式作用域限定到根节点的 CSS 类名上，然后通过皮肤（OOCSS）或者子模块（SMACSS）进行修改。除了 SMACSS 把代码划分类别之外，两者之间最显著的差异是使用皮肤而不是子模块，以及带 `is` 前缀的状态类名。

4.4.3　BEM 方法

这里同样以切换组件为例，使用 BEM（Block Element Modifier，块元素修饰符）语法写出组件代码：

```
<div class="toggle toggle--simple">
  <div class="toggle__control toggle__control--active">
    <h2 class="toggle__title">Title 1</h2>
    </div>

    <div class="toggle__details toggle__details--active">
      ...
    </div>
    ...
  </dl>
```

BEM 是我们要看的第三个方法，是 SMACSS 的另一个方面。BEM 只是一个 CSS 类名命名规则。它不涉及如何书写你的 CSS 的结构，而只是建议每个元素都添加带有如下内容的 CSS 类名。

块名

　所属组件的名称。

元素

　元素在块里面的名称。

修饰符

　任何与块或元素相关联的修饰符。

BEM 使用非常简洁的约定来创建 CSS 类名，而这些字符串可能会相当长。元素名加在双下划线后（例如 `toggle__details`），修饰符加在双横杠后（如 `toggle__details--active`）。这里的 `details` 是元素，`active` 是修饰符，这个约定使得 CSS 类名非常清晰。使用双横杠是为了避免块名被混淆为修饰符。

这种方法在 OOCSS 或 SMACSS 里使用的好处是，每一个 CSS 类名都详细地描述了它实现了什么。代码中没有 `open` 或者 `is-active` 这样只在特定背景下才能理解的 CSS 类名。如果单独看 `open` 和 `is-active` 这两个名字，我们并不知道它们的含义是什么。虽然 BEM 方法看起来很累赘、很冗余 ，但是当看到一个 `toggle__details--active` 的 CSS 类名，我们就知道它是表示：这个元素的名称是 `details`，位置在 `toggle` 组件里，状态为激活。

4.5　选择适合的方案

当然，最重要的还是要找到一个适合的解决方案。不要因为一套规范很流行或者别的团队正在使用就选择它。这三种方法都提供了类似的工具，并且以相近的方式在系统中使用。

在 Red Hat 网站中，我们使用了 SMACSS 和 BEM 相混合的方案，这一点将在第 7 章中详细讨论。不要害怕尝试、混合或者创造出独一无二的方案！你要做的是理解这些方案背后的原则，说出为什么你的方案能够解决项目面临的挑战，并且确保团队乐意使用一个统一的方案。如果你决定使用 OOSMABEM，大胆去做吧！我期待能够读到关于它的东西。

第 5 章
CSS

这个行业最好的事情之一是：我们可以随意地和其他开发人员坐下来，喝点咖啡聊聊工作。这似乎无关紧要，但我告诉你，这非常重要！我们在一个以标准开放、软件开源、信息自由、学习便捷为基础的行业中工作，使用的工具和技术变化之快令人难以置信，但正是这样一个开放的行业氛围使得我们能够紧跟趋势。这可能会让你感到惊讶，但在一些领域里，你永远不会，也不可能在一些商业场合坐下来与同行讨论具体工作内容，除非他们付钱让你这么做。在那些领域中，每一个知识、每一个技巧、每一个预设、每一个宏命令、每一个文档，以及每一个捷径都可能被挂牌出售，而你最不想做的事情就是跟潜在的竞争对手一起坦诚地交换那些信息。

但 Web 领域不同，我们正是依赖分享知识才变得更加强大。我们发表博客文章、录制视频教程、创建公共代码仓库、在 Stack Overflow 上发表帖子、回答 IRC 的问题，以及发布代码到 CodePens、Gists、Sassmeisters、JSbins 和 Pastebins 上，所有的这些都是为了让其他人可以学习到我们所了解的东西。喝杯咖啡，讨论现行的 Web 开发实践和新的 CSS 框架，这就是我们如何分享知识以及如何在这个行业中学习的最基本表现。

因此，在我们从事的行业中，邀请同行一起出去喝咖啡不仅是可接受的，而且还会被当成一种有意义的行为！这些咖啡会帮助我们学到新的事物，获得新的商务联系，找到新的工作，甚至交到很好的朋友。可以说，多年以来的喝咖啡、喝酒、喝茶或者红茶菌的经历，使我相信一起喝东西时的交流是我们这个行业宝贵的无形资产。

坚持用咖啡或者酒精促进谈话的好处在于，我们会不断了解人们对什么感兴趣，以及人们在朝着什么方向努力。这对我来说很重要，不是因为我依赖别人来知道下一步该做什么，而是因为我能够验证我所学习的东西以及我自己的发现。每一年我都会回顾我的 CSS 方法

是如何发展的。所以，无论我最新痴迷的是预处理、构建工具、样式驱动的设计，还是基于组件的设计系统，我都会因为和别人有完全相同的发现而感到很兴奋。

我现在编写的 CSS 跟三年前的大相径庭。每建设一个网站，我都会从中学到新的东西，并尝试应用到下一个项目中。这是一个逐步发展的过程，我把学到的东西和读到的东西集聚到一起，并尝试用改进的方案解决我所面临的特定问题。每一次迭代都提升了我的技术，也大大加深了我对可扩展性和可维护性设计系统的理解。

每次迭代都会带来一些激动人心的讨论，也许是与各种各样的同事、业内同行，也许是与一些参会者。我会因为每次谈话中的共鸣而感到狂喜，比如发现我并非唯一使用数据属性或可选样式来处理组件变量的人！

作为一名前端架构师，你可能不需要知道不起眼的 Opera Mini 版本上的每一个细微的 CSS bug，但是你必须理解 CSS 的主要发展趋势，并能制定会让团队获得成功的计划。如果你不了解目前 CSS 的发展趋势，或者没有快速领悟，那么尝试和其他开发人员多喝几杯咖啡吧，或者在你下一次参加行业会议时，多花一些时间与参会的同行进行社交互动。但是在你做这些事之前，请阅读下面的章节，你会了解几年前 CSS 技术的情况，当时写的 CSS 为什么不生效，以及现在的 CSS 技术发展成什么样。

5.1　特性之争与继承之痛

过去几年，我们仍然在处理我在前一章中描述的 100% 动态标记或 100% 静态标记。不论是哪一种，我们几乎总是从全局作用域开始开发，一层一层增加细节。我们从使用通用样式开始，比如页首和段落的标签，然后给页面里的各个部分应用具体的样式：

```
<body>
  <div class="main">
    <h2>I'm a Header</h2>
  </div>
  <div id="sidebar">
    <h2>I'm a Sidebar Header</h2>
  </div>
</body>

<style>
h2 {
  font-size: 24px;
  color: red;
}

#sidebar h2 {
  font-size: 20px;
  background: red;
  color: white;
}
</style>
```

现在侧边栏的 H2 是白色字体红色背景，除此之外的每一个 H2 都是红色字体。

这个概念很容易理解，如果你是印刷行业出身，会百感交集。每次你把一个 H2 放在侧边栏，它会被应用相同的样式，直到我们需要用到一个在侧边栏的日历组件，又要求 H2 显示为和页首一样（红色字体且不填充背景色）。但是没关系！我们可以添加另一个类名并且重写让人讨厌的侧边栏样式，对吧？

```
<body>
  <div id="main">
    <h2>I'm a Header</h2>
  </div>
  <div id="sidebar">
    <h2>I'm a Sidebar Header</h2>
    <div class="calendar">
      <h2>I'm a Calendar Header</h2>
    </div>
  </div>
</body>

<style>
h2 {
  font-size: 24px;
  color: red;
}

#sidebar h2 {
  font-size: 20px;
  background: red;
  color: white;
}

#sidebar .calendar h2 {
  background: none;
  color: red;
}
</style>
```

这种方法的问题很多。

选择器优先级

无论你处理带 ID 的标签还是长选择器，重写一个选择器时，总是需要注意它的优先级。

颜色重置

要恢复到原来的 H2 颜色，我们必须再次指定样式，并且要覆盖当前的背景颜色。

位置依赖

现在我们的日历样式依赖于侧边栏的样式。如果将日历移到页首或者页尾，样式将会改变。

多重继承

现在这个 H2 的样式来源多达三个，这意味着只要改变主体或侧边栏的样式，都会影响

到日历的呈现。

深层嵌套

日历控件里的日历条目可能还有其他的 H2，而它们也需要一个更具体的选择器，这样一来，H2 的样式来源就增加到了四个。

5.2 一种现代的、模块化的方法

我们在第 4 章讨论 HTML 时就涉及了一些现代模块化原则，大多数 CSS 框架用它们来解决刚才描述的方案中出现的问题。虽然 OOCSS、SMACSS 和 BEM 对使用什么样的标记有着不同的观点，但它们中的每一个都提供了如何写 CSS 的建议，无论你采取了哪一个方案，这些建议都很有价值。让我们来快速看一下这些关键原则，并且了解它们是如何帮助我们解决前面遇到的问题的。

OOCSS 带来分离容器和内容的思想，我们学会不再使用位置作为样式的限定词。你完全可以在网站上放一个侧边栏，并且给这个侧边栏使用任何你喜欢的样式，但是，侧边栏的样式不应该影响侧边栏的内容。`#sidebar h2` 意味着，放在侧边栏的每一个 H2 元素，要么接受这个样式，要么就重写来避免使用这个样式。而 `.my-sidebar-widget-heading` 意味着样式只限定于这一个标题，完全不会影响其他标题。

SMACSS 给我们带来把布局和组件分离到不同文件夹的思想，进一步将侧边栏的角色和日历模块划分开。现在我们只是定义了侧边栏的角色是布局，甚至不允许元素样式在那部分 Sass 语法的代码里出现。如果你要在侧边栏里放一些代码，并且向它们添加样式，那么这些元素需要是某个组件的一部分，并且需要在组件的文件夹里定义。

虽然严格来说，BEM 并不算是一种 CSS 方法论，但它让我们知道，给标记中每个 CSS 类名一个独一无二的标识是有价值的。这是因为这样会使每个 BEM 风格的 CSS 类名都可以对应到某一组独属于该元素的 CSS 属性，而不会随着具体情境或选择器的使用而变化：

```
<body>
  <div class="main">
    <h2 class="content__title">"I'm a Header"</h2>
  </div>
  <div class="sidebar">
    <h2 class="content__title--reversed">
  "I'm a Sidebar Header"
    </h2>
    <div class="calendar">
      <h2 class="calendar__title">"I'm a Calendar Header"</h2>
    </div>
  </div>
</body>

<style>
```

```
/* 组件文件夹 */
.content__title {
  font-size: 24px;
  color: red;
}

.content__title--reversed {
  font-size: 20px;
  background: red;
  color: white;
}
.calendar__title {
  font-size: 20px;
  color: red;
}

/* 布局文件夹 */
.main {
  float: left;
  ...
}
.sidebar {
  float: right;
  ...
}
</style>
```

这就解决了刚才的由于依赖位置而造成 CSS 样式混乱的问题。

选择器优先级

把 ID 选择器改成 CSS 类名选择器是一个很好的开始，这样可以停止 CSS 优先级之间的冲突问题，让每一个选择器的权重扁平化成“1”，我们就不再需要利用优先级较量出“胜利者”来决定样式。

颜色重置

比降低权重更好的方法是对每一个元素使用唯一的选择器。这样你的模块样式就不再会与侧边栏样式或者页面通用样式冲突了。

位置依赖

去掉布局文件中的样式代码之后，我们就不用再担心因为把日历组件移出侧边栏而造成样式改变了。

多重继承

每个标题都有了自己唯一的 CSS 类名之后，我们就可以任意修改其中的某个样式而不会影响其他标题了。如果你想改变多个选择器对应的样式，可以使用预处理器变量、混入（mixin）或继承来帮你做。

深层嵌套

即使在日历的层级上，我们也仍然没有给 H2 标签应用任何样式。因此再给日历中的新 H2 添加样式时，就不需要重写通用样式、侧边栏样式或者日历的头部样式了。

5.3 其他有助益的原则

5.3.1 单一职责原则

在建立 Red Hat 网站的过程中，我发现了一些对规划 CSS 框架有极大帮助的东西。

*单一责任原则*规定你创建的所有东西必须有单一的、高度聚焦的理由。你应用到某个选择器里的样式应该是为了单一目的而创建的，并且能够很好地实现这个目标。

这并不意味着你应该为 `padding-10`、`font-size-20` 和 `color-green` 设置单独的 CSS 类名。我们关注的是样式适用在哪些地方，而不是这些样式本身。让我们来看下面的例子：

```
<div class="calendar">
  <h2 class="primary-header">This Is a Calendar Header</h2>
</div>

<div class="blog">
  <h2 class="primary-header">This Is a Blog Header</h2>
</div>

.primary-header {
  color: red;
  font-size: 2em;
}
```

虽然上面的例子看似相当有效，但是它显然不符合我们的单一责任原则。`.primary-header` 这个 CSS 类名被应用于页面上不止一个不相关的元素上。现在，`primary-header` 的“责任”是既负责日历的标题也负责博客的标题，这意味着对博客的标题做任何更改也会影响到日历的标题，除非你像下面这么写：

```
<div class="calendar">
  <h2 class="primary-header">This Is a Calendar Header</h2>
</div>

<div class="blog">
  <h2 class="primary-header">This Is a Blog Header</h2>
</div>

.primary-header {
  color: red;
  font-size: 2em;
}

.blog .primary-header {
  font-size: 2.4em;
}
```

这种方法虽然在短期内有效，但是也使我们再次面临本章开头所提到的那几个问题。这个新的标题样式现在取决于元素位置，具有多重继承性，并且引发了“最高优先级”争夺赛。

针对这个问题，一个更加可持续的方法是让每个 CSS 类名都有单一的、有重点的任务：

```
<div class="calendar">
  <h2 class="calendar-header">This Is a Calendar Header</h2>
</div>

<div class="blog">
  <h2 class="blog-header">This Is a Blog Header</h2>
</div>
.calendar-header {
  color: red;
  font-size: 2em;
}

.blog-header {
  color: red;
  font-size: 2.4em;
}
```

虽然这种方法确实会导致一些代码重复（红色字体定义了两次），但是它的可持续性带来的好处大大超过代码重复的任何坏处。这里多出来的代码对网页大小的增加而言微不足道（gzip 喜欢重复的内容），而且由于博客标题不一定一直保持红色，如果整个项目强制执行单一责任原则，就能够确保在进一步改变时，我们可以毫不费力地完成，并且也不需要回顾之前的代码。

5.3.2　单一样式来源

单一样式来源的方法将单一责任理论应用到更深层次，不仅每个 CSS 类名被创建为单一用途，而且每个标签的样式也只有单一的来源。在一个模块化设计中，任何组件的设计必须由组件本身决定，而不应该被它的父类名限制。让我们看看实际应用中的情况：

```
<div class="blog">
  <h2 class="blog-header">This Is a Blog Header</h2>
  ...
  <div class="calendar">
    <h2 class="calendar-header">This Is a Calendar Header</h2>
  </div>
</div>

/* calendar.css */
.calendar-header {
  color: red;
  font-size: 2em;
}

/* blog.css */
.blog-header {
```

```
  color: red;
  font-size: 2.4em;
}

.blog .calendar-header {
  font-size: 1.6em;
}
```

写这样的样式是为了当日历在博客文章里出现时，缩小日历头部文字的字体。从设计的角度看，这无可厚非，最终日历组件会根据它在哪里而改变外观。我们称这种带条件的样式为“上下文”，已被广泛应用到我的设计系统里。

这种方法的主要问题是，定义缩小字体的样式代码在博客组件的文件中，而不是单一地存在于日历组件文件。在这种情况下，样式散落在多个组件文件里，导致很难预料某个组件放到页面中会长什么模样。为了缓解这个问题，我建议把带上下文的样式移到日历模块代码中：

```
<div class="blog">
  <h2 class="blog-header">This Is a Blog Header</h2>
  ...
  <div class="calendar">
    <h2 class="calendar-header">This Is a Calendar Header</h2>
  </div>
</div>

/* calendar.css */
.calendar-header {
  color: red;
  font-size: 2em;
}

.blog .calendar-header {
  font-size: 1.6em;
}

/* blog.css */
.blog-header {
  color: red;
  font-size: 2.4em;
}
```

通过这种方法，当日历头部出现在博客文章里时，我们仍然能够缩小它的字体，而且如果把所有带 `calendar-header` 上下文的样式放到日历文件中，我们可以看到日历头部所有可能的变动都放在一起。这使得更新日历模块更容易（因为我们知道所有变动情况），也使得我们能为每一个变动创建适当的测试覆盖。

5.3.3　组件修饰符

虽然单一样式来源的方法确实可以让代码变得更清晰，但是如果要跟踪多个不同的上下文还是很困难的。如果你发现日历头部需要在数十个上下文中显示较小的字体，那么是时候

改用修饰符类名了。

组件修饰符（又称皮肤或者子组件，取决于你所赞同的方法论）让你能够定义一个组件在多个不同情况下的多种变化。它的工作方式和单一样式的来源方法非常相似，但是修饰符类名的属性不再是父组件，而是组件本身的一部分：

```
<div class="blog">
  <h2 class="blog-header">This Is a Blog Header</h2>
  ...
  <div class="calendar calendar--nested">
    <h2 class="calendar-header">This Is a Calendar Header</h2>
  </div>
</div>

/* calendar.css */
.calendar-header {
  color: red;
  font-size: 2em;
}

.calendar--nested .calendar-header {
  font-size: 1.6em;
}

/* blog.css */
.blog-header {
  color: red;
  font-size: 2.4em;
}
```

在前面的例子中，我们使用传统的 BEM 语法创建了 `calendar--nested` 修饰符。这个 CSS 类名单独使用时什么都不做，而当它被应用到日历组件上时，组件里的元素就能拿它做局部上下文并改变外观。

我们能够用这种方法把修改过的日历皮肤使用到任何地方，从而得到更小的字体（以及想要的其他改变）。这保证了所有组件的变动都在一个文件里，而且能用到任何需要的地方（或者不使用它们），而不依赖于不确定的父节点 CSS 类名。

5.4 小结

能够帮助你创建可维护代码库的 CSS 技巧有很多，本章涉及的只是冰山一角，包括的内容如下：

- 分离容器和内容
- 区分布局与组件的角色和职责
- 在标记上使用单一、扁平的选择器
- 使用其他原则，比如单一职责原则、单一样式来源、内容修饰符

本章和前一章列出了很多建议，任何形式、任何大小的项目都可以从中获益，但最终如何编写 CSS 仍由你和你的团队决定。我唯一的要求是，通过对这些问题的探讨，你能够加深对前端架构的思考，建立起什么是好架构和如何搭建架构的观念，并在代码评审时能够使用其中的某些原则作为分析和评论的根据。这样做可以让你巩固代码核心，使之更好地支撑起整个前端架构，并且帮助团队走向成功。

第 6 章

JavaScript

虽然很多人认为不应该把 JavaScript 纳入 Web 开发必备技能里，但它却是目前很多网站不可或缺的一部分。不论你创建的是小型作品集站点、企业门户网站还是电商网站，你难免会遇到只用 HTML 和 CSS 无法解决的问题，可能是幻灯片和图片灯箱，也可能是完整的客户端应用。

本书主要关注如何创建可扩展且可持续的设计系统，所以我会重点帮助你建立项目计划和辨别高质量代码。至于“如何创建基于 JavaScript 的 Web 应用”，就留给其他相关的书去讨论了。实际上，谈到 Web 应用，让我们先来讨论一个重要的话题：如何选择完美的框架。

6.1　选择框架

首先需要指出的是，没有哪个 JavaScript 框架是完美的。本章也不会告诉你应该选择 AngularJS 还是 ReactJS，更不会尝试列举出所有的可选框架，因为当这本书出版时，这些框架很多都会变得过时，而且这种列表必定会遗漏那些最新的、最流行的框架。

我要告诉你的是：你很可能完全不需要用任何框架。首先需要决定的是选择哪些工具来实现目标，而不是选择哪些框架和插件。

这一点不仅适用于 JavaScript 框架，也适用于 CMS、MVC 和 CSS 框架。有很多成功的网站也只不过是静态网站生成器，加上少量手动创建的 Sass 文件和几十个 JavaScript 函数。

首先假设我们没有使用 Drupal 或 WordPress，也不需要 AngularJS 或 ReactJS，并且可以自己手动编写所有的样式。然后对照项目需求，在处理用户验证、页面版本管理或者 API 大

数据传输这类问题之前，看哪些需求是能先满足的。当遇到难以手动实现的需求，并且已经有开源项目或软件可以解决问题时，再开始评估第三方工具。

用精简的方案做项目，而不是一开始就准备一大套工具和大规模的启始页，除非增加复杂度和代码体积利大于弊，否则不要轻易放弃精简方案。

6.2 维护整洁的JavaScript代码

即使平时做的项目都比较简单，最复杂的情况也不过是使用 jQuery 和插件，如果能针对“如何编写 JavaScript”创建标准，你仍然能从中获益。你会发现，如果没有某种期望和规范，你的 JavaScript 文件会像断线的风筝一样不受控制，代码也难以测试和重构。作为前端架构师，在给 JavaScript 代码评审制定标准的时候，你可以参考下面概括的一些规范。

6.2.1 保持代码整洁

JavaScript 是一种脚本语言，这跟 HTML 和 CSS 不同。如果你忘记闭合一个 HTML 标签或者写了无效的 CSS，最坏的情况不过是页面上出现了一些小缺陷。如果你在 JavaScript 代码里添加了太多的逗号或者忘记闭合大括号，整个网站都有可能崩溃。

由于编写恰当的 JavaScript 非常关键，最好在项目中结合单元测试使用一些格式 / 错误提示工具。如果在开发流程里运用其中一条，这不仅有助于发现导致崩溃的代码，而且能帮助你执行关于代码格式甚至是代码编写的规范。

JS Hint（http://jshint.com/）是这些工具中一个很好的例子。它能够在你的文本编辑器里使用，一旦你犯错，它就会即时标记出来。你甚至可以把它用在构建系统中，这样如果有任何代码不符合规范，将无法通过测试。

这里有几条可以使用 JS Hint 检查的规则：

- 强制使用 `===` 和 `!==` 代替 `==` 和 `!=`
- 限制代码块嵌套深度
- 限制函数的参数数量
- 如果函数重复定义，发出警告
- 如果变量创建后未被使用，发出警告

6.2.2 创造可复用的函数

依照写 jQuery 代码的方式，我们的代码常常以 CSS 选择器（名词）开始，紧接着是执行一连串函数（动词）。这正是人类大脑的工作方式，我们首先想到需要锁定的事物，然后考虑对它做什么。虽然这使代码对人们来说可读性很强，但可复用性却并不高。来看看我

们给 CMS 创造的警告类名：

```
$('.red-alert')
   .css('background-color', "red")
   .on('click', function() {
    console.log($(this).html());
});

$('.yellow-alert')
   .css('background-color', "yellow")
   .on('click', function() {
    console.log($(this).html());
});
```

这段代码很短，也很容易读懂。我们知道 `.redalert` 给元素添加了红色背景，然后绑定事件函数，点击元素后，它的内容会被打印到控制台上。

我们也知道 `.yellow-alert` 给元素添加了黄色背景，并绑定了同样的事件函数。

对于只有两个类名的情况，这可能还比较好用，但是这样的代码是不可复用的。如果想创建更多的警告颜色，我们需要复制其中一个，然后改变类名和颜色。而且，如果因为某些原因创建了十多块这样的代码，当我们想回头调整点击函数或者添加字体颜色去搭配背景色时，可能需要改几十次才能完成。

相比写一系列描述句一样的代码，把代码拆分成小块、可复用函数可以帮我们创建出更好的系统。看看以下替代方案：

```
$.fn.log_text_on_click = function() {
  this.on('click', function() {
    console.log($(this).html());
  });
  return this;
};

$.fn.add_background = function(color) {
  this.css('background-color', color);
  return this;
}

$('.red-alert').add_background("red").log_text_on_click();
$('.yellow-alert').add_background("yellow").log_text_on_click();
```

这个方案虽然需要多写几行代码，但有以下优点：

- 现在有了清晰地说明用途的函数
- 如果需要创建新的 `.green-alert` 类名，只需要修改定义好的 `add_background()` 和 `log_text_on_click` 函数
- 如果需要将 `console.log($(this).html());` 改成 `console.log($(this).text());`，只需要在一个位置修改，而不是多个位置

- 可以在项目里的很多地方复用这两个函数

就像 Sass 的混入（mixin）写法比原生 CSS 有更多好处，把代码拆分为可复用的函数，我们的代码将变得更加清晰、精简、灵活和可测。

6.3 小结

JavaScript 仅仅用一个章节言之不尽，要讲完这个话题需要写很多本书。这里概括的技巧绝不是架构项目要考虑的全部因素，但是它们是非常好的例子，说明如果没有提前规划，代码质量会变成什么样。这并不是说开发人员不能独立写出高质量代码，而是每个开发人员对好代码的看法不一致。提前制定代码标准是确保代码评审公平的唯一方法，而且可以让开发人员更清楚他们应该如何编写代码。

第 7 章

Red Hat代码

Redhat.com 是在 2014 年的秋天发布的，而我在当年的春天才加入这个已经进行了多年的项目，因此尽管对项目发布时的代码架构非常熟悉，我却没什么机会对它做一些大的改动。这个项目涉及的范围较广，最后期限也迫在眉睫，这导致我花了很多时间赶手头的工作，几乎没有时间思考手头的工作是不是团队真正需要的最佳方案。

最后，这个网站发布了，从各方面来看都获得了成功。用户界面友好，网站加载流畅，而且几乎没有人认为该网站不引人入胜。但我还记得那个决定性的下午，我被问到："我们的设计是如何模块化的？我们希望能把主题的一小部分和其他公司网站共享。"

我内心忍不住暗暗笑了好一会儿。你看，我非常熟悉这个项目的所有标记、JavaScript 和 CSS，我也深知这个项目完全就是模块化设计的反面教材。我们在这个主题上做了很多出色的工作，但是唯独没有考虑模块化设计的问题。

7.1 过多的依赖

如果有人想渲染我们样式中的某一块内容，首先需要加载以下内容。

Bootstrap CSS：114KB（未压缩）

其实网站本身没怎么调用 Bootstrap 库的代码，但我们编写所有 CSS 的前提条件都是 Bootstrap 已经重置了基本样式，所以一旦把这些样式拿走，整个页面就不能正常使用了。

核心的网站CSS：500KB

虽然一般来说，每块内容都有一个单独的关联文件，但这个文件绝不是这块内容的单一样式来源。样式不仅来源位置多样，并且常常基于位置和页面的类被覆盖重写。

7.2 严重的位置依赖问题

这个项目所使用的标记命名方法都是按照内容块的层级顺序自上而下添加的。我们有几种不同的内容块类型，大部分样式都有固定的作用域，只能在某个内容块内重复使用。下面是一个极端的例子，在广告图版块里有一个 H3 标签：

```
.about-contact .hero1 .container >
section.features-quarter >
section.f-contact h3
```

这个样式不仅被局限在一个页面里（`about-contact`），而且我们还需要确认 `features-quarter` 部分是容器标签的直接后代，这样才不会不小心把样式添加到其他部分的元素的后代上。这种自上而下的样式命名方法意味着，每次修改我们都要写一个更长、更具体的选择器。同时，因为标记顺序极为严格，每块内容都很难重排或替换。

当然，我们可以抽出一个单独的组件，并把它需要的所有样式合并到一个单独的文件里，但是这么做基本意味着完全重做这个组件里的 Sass 文件，而且这么做也并没有真的实现标记的模块化。

于是，当我被问及我们的设计是如何模块化的而且能不能把样式分享给其他部门时，我只能说，如果要分享某部分，我们就需要重写那部分的标记和 CSS，同时还得更新网站的标记。

我确信，这样与最初设计方案背道而驰的提议，一定会使我的笑声传出这个房间。因此你就能理解，当他们居然不仅同意了，而且在新网站代码被冻结的情况下还给我们几周时间去构建新系统的时候，我为什么如此惊讶了。所以，目前我们有了完整的设计、一个非常能干的开发团队，还可以全权创建全新的模块化、可扩展和可持续的设计系统。建好之后，新系统将成为一个有名的高负载的动态网站。我只想知道，我们什么时候可以开始？！

7.3 设计分解

正如之前所说的，我们原来采用的是自上而下的设计方法。内容、外观和布局方式完全由内容版块决定。我们有商标墙、广告图版块、感言版块、博客广告，以及其他你想要的版块。每一个版块里，都有自己的 Sass 文件，并且这部分样式只能在所属版块名称下生效。

在某种程度上，这种方式还是相当好用的。通常情况下，你知道去哪里更新特定版块的样式。但问题是，当我们需要采用新的设计或者布局时，就不得不一再新建版块。我们可能有完善的视觉语言，但是从来没有“翻译”成一个可以模块化生成设计样式的系统。

所以，我们的首要任务是把设计分解成尽可能小的单位。我们知道，一旦有了建造设计系统的基本构件，就可以创造任何视觉语言所能传达的东西。所以第一步就是细看我们的设

计，并且把它分解为可复用的布局模式。

通过对比几个最常用的内容块类型，我们注意到它们大多数都共用了几种常见的布局模式（见图 7-1）。一般它们有带侧边栏的内容，有同样大小的列内容，有每行五张的图片集和图标集，还有带一点内边距和一个背景图的黑色卡片和白色卡片。

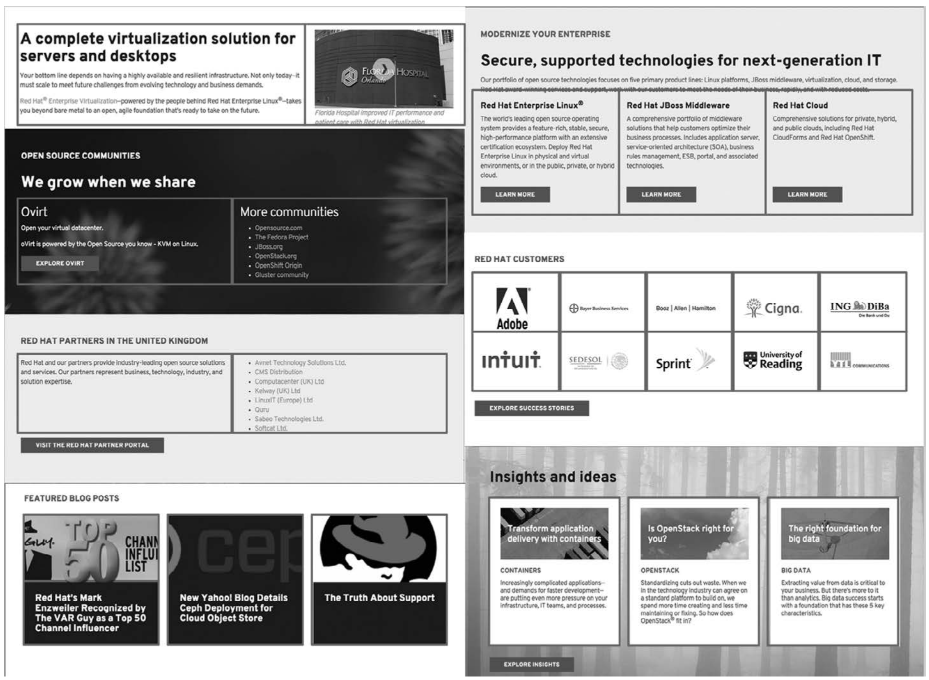


图 7-1：我们的设计系统的多种布局

我们很快意识到，如果能创造一种方法复用这些简单的布局模式，我们就可以为网站里的每一个内容块生成布局。我们不再需要为商标墙和 insights 版块分别创造单独的布局。我们可以新建有很多可选样式的版块，还可以专门写一个卡片布局，只是加一点内边距，再为内容添加黑色或白色的背景。

7.4 组件分类

我们用之前分解布局的方法分解每个布局的内容，然后意识到只需要写一些好用的组件小工具就可以模块化地生成大部分内容了。

相比布局，组件只描述了一小块内容的视觉外观。组件很灵活，没有任何背景、宽度、内边距、外边距等设置。

实践证明，布局和组件之间的关系有着强大的威力，我们把三个组件放在一个三列布局中，不需要写代码就能让它们看起来浑然一体。每个组件都会填满所在列的列宽，第一列里的引用文字和第二列博客入口文字以及第三列的图片都自动水平对齐了（见图 7-2）。



图 7-2：因为组件都没有上边距，所以图中三个不同的组件对齐了

当意识到这个强大的系统是基于一些简单也非常重要的规则和关系时，我们把它们整理成更加正式的形式。我们希望确保当新的布局和组件引入到系统中时，每一次的合并请求都符合一系列的规则。我们称这些规则为“BB 鸟规则”。

7.5　BB鸟规则

在思考这些新规则之前，我的 Twitter 被 Chuck Jones 的故事和他的九条规则的列表刷爆了，据说这些规则是写给 BB 鸟动画片的作家和动画师的。在 BB 鸟的世界里，这个列表设定了一些规则，比如“BB 鸟不能伤害歪心狼，除非听到了‘哔哔’”“永远没有对话，除非听到了‘哔哔’”。这些规则帮助每集动画的作者和动画师创造一致的、有凝聚力的作品。

不管这个故事是否真实，我想给自己的团队塑造的就是这样一致的、有凝聚力的状态。为了避免写出类似“迅猛的 BB 鸟撞到了歪心狼的下巴，接着听到歪心狼喊叫‘你抓不到我，傻瓜！’”的情节，我想到的唯一办法就是从中提取出管理设计系统的规则，得到一个精简的、好用的 BB 鸟规则。

7.6　编写你自己的规则

我刚开始编写 BB 鸟规则时，规则数目是现在的两倍。而当规则数目达到了两位数，我发现每写一条规则，我都会想出两条、三条甚至更多的规则。

等做完这么多规则，估计花都谢了。我意识到我根本不是在制定规则，而是在写整个系统的说明文档，然而问题是我已经写过文档了。

事实上，我并不需要为歪心狼受伤倒下的结局，详细地描述歪心狼如何从最大的分类开始

查找、选购商品，然后几秒钟后商品就出现在他的信箱里。我只需要规定“没有外力可以伤害歪心狼，除非他很笨或是他买的高科技产品出故障”。

我需要的是精简的、固定的规则列表，而不是一个完全展开的详细说明。很显然，我还需要一些规范来约束规则的制定。

现在我知道，你大概在想“这要么是过分的分解，要么是愚蠢的尝试”。但是就像一个好看的导航条会有它独特的设计和规则，一个扎实的数据写入模式也是按照父模式的规范写的，我们的 BB 鸟规则也需要遵循它自己的管理规范。

下面就是我们制定 BB 鸟规则时需遵循的规范：

- 只包含不可变的规则，而不是笼统的说明
- 总是把规则提炼成最简单的表达
- 总是首先说明规则是什么，再说明“如果不这样，那么会如何”
- 每个规则必须包含以下词中的一个——总是、永远不要、只有、每一个、不要、要

使用这些规则可以帮助我们避免写了很多内容却始终说不到点子上。聪明的读者已经发现这四条规则和 BB 鸟的设计原则是相吻合的。

在对这个列表进行了改写、重写、删除等操作后，我们最后制定了自己的设计系统的规则列表。

- 永远不要给布局的子内容强加内边距和元素样式。布局只关注垂直对齐、水平对齐和文字间距。
- 主题和别的数据属性值永远不要强制改变外观；它们必须保持布局、组件和元素可以应用于其上。
- 组件总是贴着它的父容器的四个边，元素都没有上外边距和左外边距，所有的最后节点（最右边和最下边的节点）的外边距都会被清除。
- 组件本身永远不要添加背景、宽度、浮动、内边距和外边距的样式，组件样式是组件内元素的样式。
- 每个元素都有且只有一个唯一的且作用域只在组件内的 CSS 类名。所有的样式都是直接应用到这个选择器上，并且只有上下文和主题能修改元素的样式。
- 永远不要在元素上使用上外边距，第一个元素总是贴着它所在组件的顶部。
- JavaScript 永远不要绑定任何元素的 CSS 类名，选中元素通过数据属性实现。

这些规则不仅覆盖了布局和组件的特定关系，也覆盖了设计系统的其他部分，包括主题、元素和 JavaScript。接下来，我们深入到我们做过的一些关于 HTML 和 CSS 的更有趣的规定。

7.7 每个标签指定唯一的选择器

过去，我花费了太多时间创造通用的、万能的、可以应用到任意元素上的 CSS 类。但是当项目发展壮大时，你才会明白维护它们有多难。因为这些 CSS 类具有通用性，改动了样式有可能会影响到很多地方，所以创造新的 CSS 类往往比更新原有的 CSS 类更容易。因此，给每一个元素创造单一的、唯一的、扁平的 CSS 类是我最想做的事情之一。你可以在第 6 章读到很多与此相关的内容，而这个原则是我们履行“每个元素都有且只有一个唯一的、作用域只在组件内的 CSS 类名”规则的关键。

7.7.1 单一责任原则

在某些领域，CSS 的单一责任原则意味着每个 CSS 类都有一个简单的、高度聚焦的责任，所以在某个场景下，用一个 CSS 类来设置元素的盒模型的属性，另一个设置排版，还有一个设置颜色和背景。

对于设计系统和规则列表来说，这个单一责任原则意味着我们创建的每一个 CSS 类都用于单一的目的和单一的位置。因此，如果我们要改变 `.rh-standard-band-title` 的样式，我相信这对网站唯一的影响是 `rh-standard-band` 里面的标题外观会改变。

这也意味着如果删除 `rh-stand-band` 的样式，即使我们删除所有关联的 CSS，也不用担心因为被“强制依赖”劫持而影响其他组件的样式。我会确保每个 CSS 类只为一个目的而创建，也只会用于这一个目的，因为我们不想滥用级联的选择器（https://www.phase2technology.com/blog/used-and-abused-css-inheritance-and-our-misuse-of-the-cascade/），参见图 7-3。

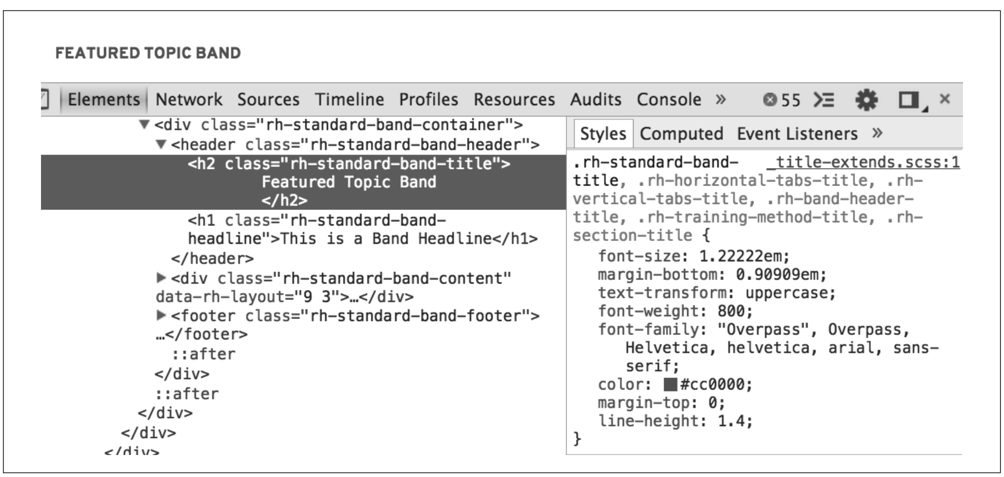


图 7-3：每个 CSS 类只为单一目的而创建

7.7.2　样式只有单一的来源

一旦页面的每个元素都只有唯一的 CSS 类，我们就可以非常自信地改变 `.rh-standard-band-title` 而不用担心影响系统的其他部分。但是 `.rhstandard-band-title` 不会被别的东西影响又意味着什么呢？这就是给每个组件和每个元素的样式保持单一来源非常重要的原因。这意味着标签不依赖于 H2、header H2 或者绑定的 Sass 文件之外的其他任何选择器。

这并不是说我们的标题不能被外力所改变，而是说，修饰符与上下文对标签样式的任何改变都会和标签的原始样式定义在同一个地方，而不会分散在不同文件。因此，当我需要使用类似 `.some-context .rh-standard-band-title` 的样式时，我总是要把样式写在标准绑定 Sass 文件里，永远不会写在别的地方。

7.7.3　可选的修饰符

我提到过，我对在组件上使用修饰符没有异议，但是每一个单独的样式变更，都需要做成可选。这意味着如果我定义了一个修饰符 A 给组件 B，那么修饰符 A 不会影响任何组件 C，除非 C 也选用了这个修饰符。

在深入探讨例子之前，我解释一下我做过的一个关于修饰符和上下文的架构选择。虽然 BEM、SMACSS 和 OOCSS 都有关于修饰符、主题、皮肤的规定，但它们都需要添加变更的 CSS 类到内容块或者元素上，而我却决定用一个不一样的不需要添加额外的 CSS 类的新方案。正如 Ben Frain（https://benfrain.com/oocss-atomic-css-responsive-web-design-anti-pattern/）所说的那样，“让一件东西只是它本身”。我不想让任何人混淆“某个东西”的 CSS 类和它的修饰符。所以我决定所有的修饰符都用数据属性代替，如下：

```
<div class="foo" data-bar="baz">...</div>
```

除了能够区分角色和目的，这么做的另一个好处是，CSS 类的属性都是一维的：应用状态或非应用状态。而数据属性是二维的，有数据属性本身和通过它传递的值。为了补偿 CSS 类缺少的维度，你会经常发现 CSS 类会使用命名空间来定义该标签属于哪个分组。一个数据属性有明确的命名空间，因此我们可以给它传递任何必要的值。虽然传递的值可能只是几个字符，但使用数据属性本身就能够强调组件拥有 `data-align` 属性，并且我们可以给它赋各种各样的值。

好，现在我们回到那个例子上来。

`rh-card--layout` 是用来给内容添加内边距和背景的布局工具。这个卡片（见图 7-4）有一个黑色背景，因为我们定义了 `data-rh-theme="dark"` 的属性含义为在卡片 Sass 文件里将卡片的背景色设置为黑色。就像我们定义 `data-rh-justify="center"` 的含义为将卡片内容居中一样。

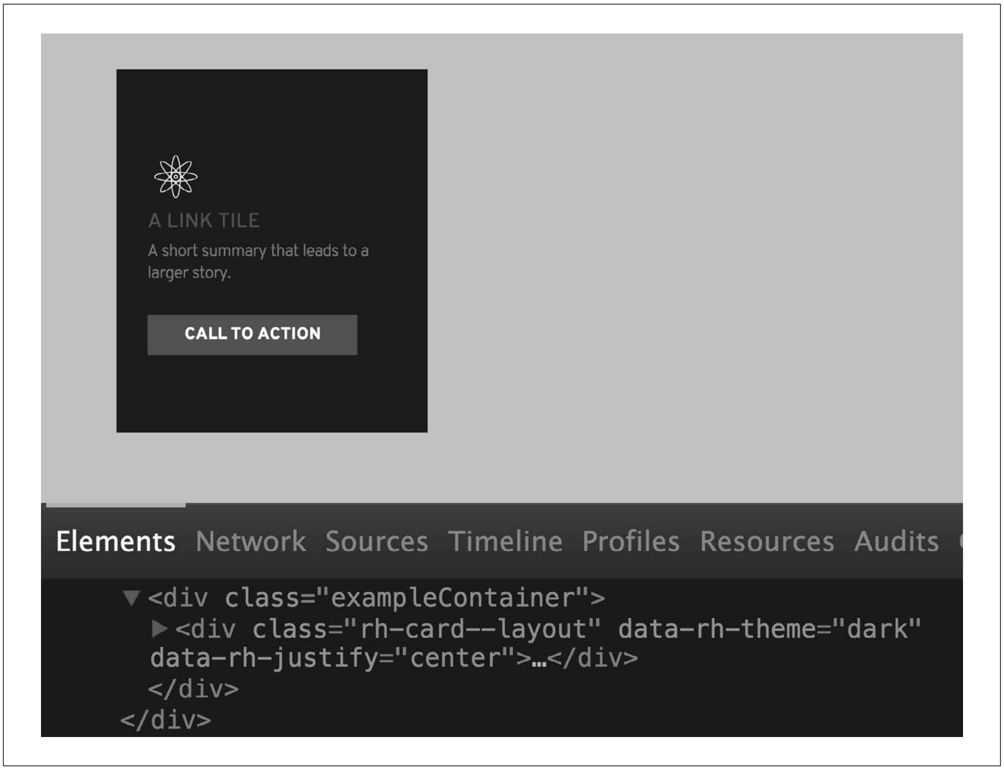


图 7-4：一个带黑色背景和深色主题的基本卡片

```
<!-- card.scss -->
.rh-card--layout {
  padding: 30px;
  &[data-rh-theme="light"]{
    background: white;
  }
  &[data-rh-theme="dark"]{
    background: black;
  }
  &[data-rh-justify="center"]{
    ...
  }
  &[data-rh-justify="top"]{
    ...
  }
  &[data-rh-justify="justify"]{
    ...
  }
}
```

除了 `center`（居中对齐），我们还指定其他对齐的值，包括 `top`（顶部对齐）和 `justify`（两端对齐），见图 7-5。

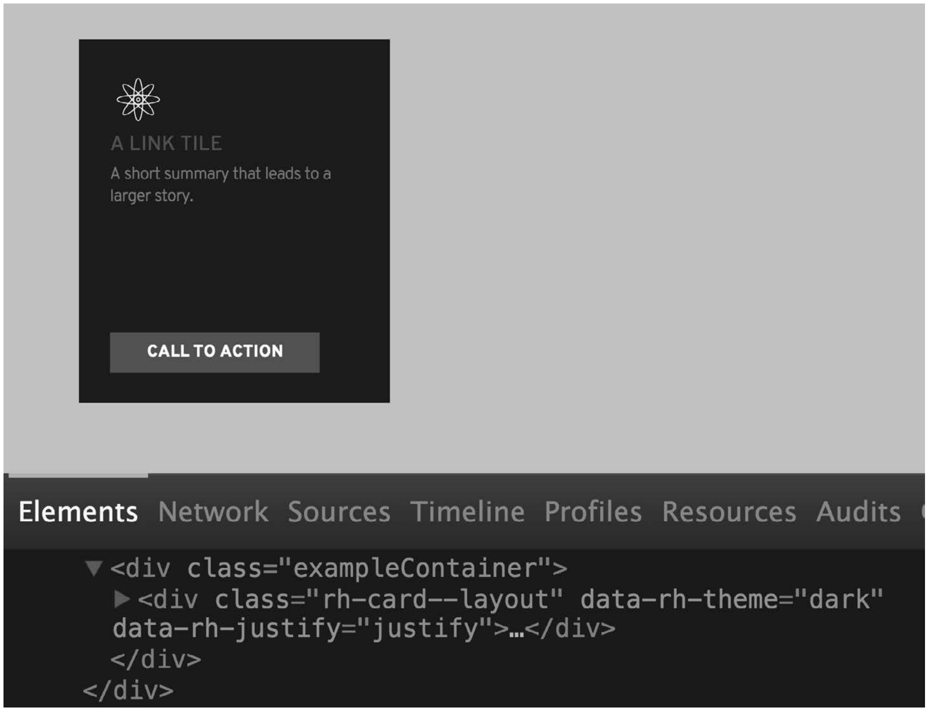


图 7-5：一个内容垂直对齐的卡片

因此，只使用数据属性的切换，我们就可以根据具体卡片的 Sass 文件的值，切换卡片的外观。

现在还有一个属性我们没有在卡片的 Sass 文件里定义，那就是对齐。通常，我们以左 / 右 / 居中对齐作为选项，通过添加 `data-rh-align` 属性给某个组件添加对齐样式，因此我们在这里有意不使用该属性。这意味着不管其他组件或布局使用了什么样的修饰符，都不会对卡片产生影响（见图 7-6）。

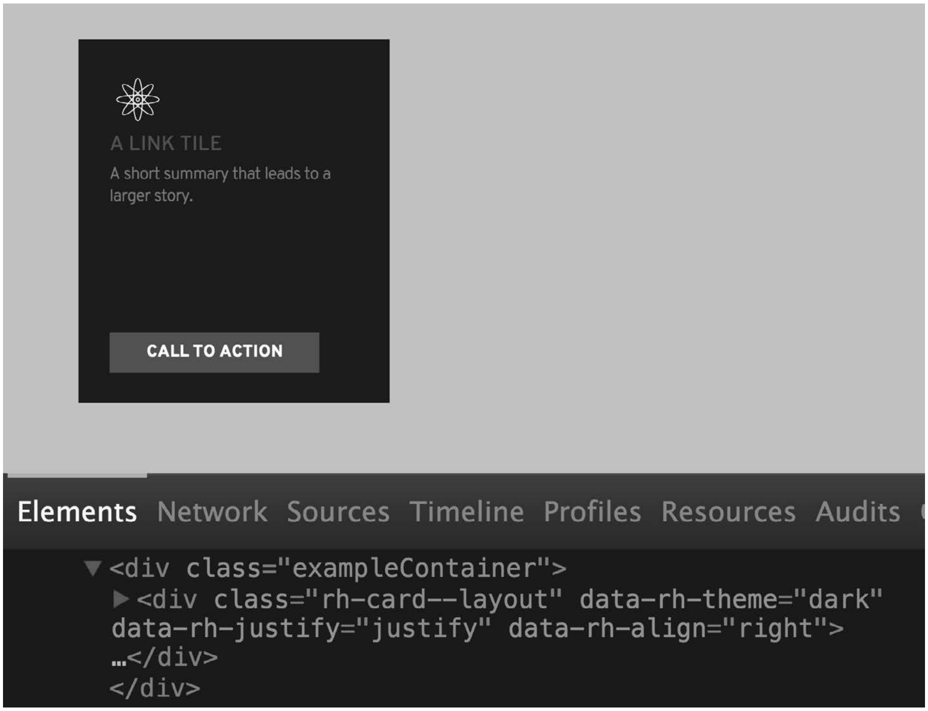


图 7-6：这个卡片不受没有明确规定的属性影响

7.7.4　可选的上下文

我们的设计系统有一个魔咒：一个组件无论放在哪里，都呈现一样的外观。但我们希望组件不仅可控并具有一致性，而且智能和灵活。这就是我们想出了上下文样式的原因。

上下文样式的含义是，组件可以根据所在的父级元素或者父级元素的某些数据属性来改变自身的表现。回到侧边栏标题的例子，我们并不是把所有的侧边栏的 H2 字体都设置为绿色，而是当我们需要 `.widget-title` 的标题在侧边栏里显示绿色的时候，我们可以做到这一点！让我们再次看看卡片的例子。

在这张卡片里，`data-rh-theme` 属性实际上既是修饰符，也是上下文。它作为修饰符把卡片的背景从黑色变成白色（见图 7-7），但是对于里面的元素来说它也是一个上下文信息。

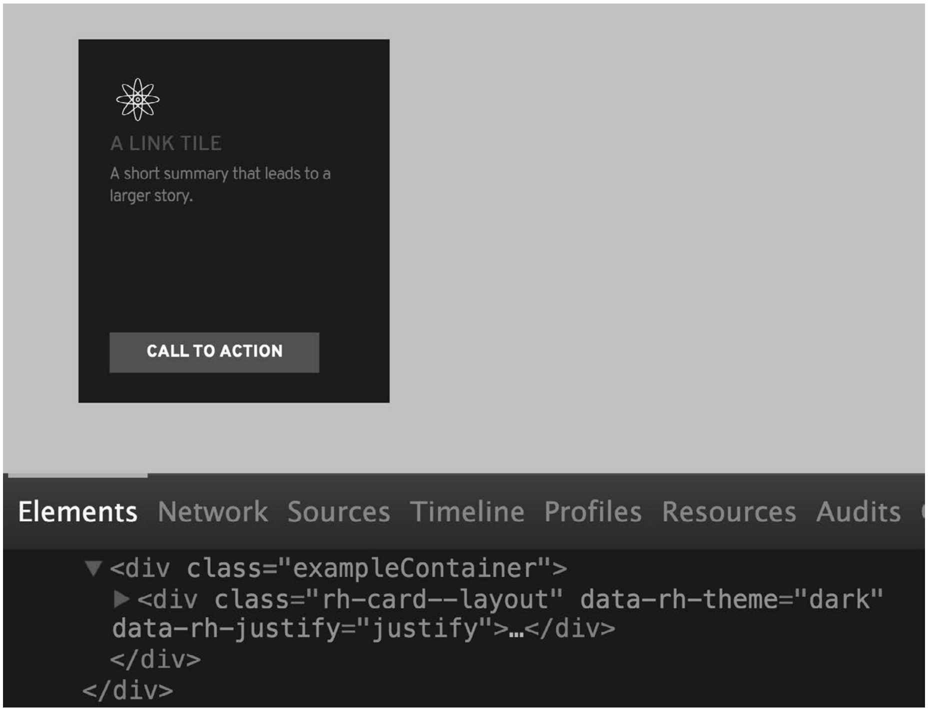


图 7-7：这个深色主题的卡片有一个白色的图标

当主题从黑色切换成白色，你可以看到不仅背景颜色变了（修饰符），图标颜色也从白色变成了黑色（见图 7-8）。但我并不是指新的主题强制把所有子节点的文字都改变了颜色（标题和按钮的颜色根本没有改变），而是这个改变对于图标来说是可选样式，可选样式的信息都会写在 `link-tile` 的 Sass 文件中（见图 7-9）。

这些可选的上下文允许我们给任何组件创建变动，而不影响组件的原始样式。这些变动都是可控的，作用域限制在组件的 CSS 类中，并且只在组件的 Sass 文件里定义。

如果让同一个修饰符和上下文影响多个组件，这的确需要我们做一些重复的工作。但随着系统规模日渐扩展，我从来没有后悔过。不仅因为通过混入（mixin）和扩展（extend）让复用修饰符和上下文变得更简单，而且组件变动的数量有限，这帮助我们避免了找错难的问题，也帮助我们提高了创造更全面的视觉还原的能力。因为所有修饰符和上下文都定义在 Sass 文件里，所以我们可以查看任意组件，确切地知悉其可能的视觉变化。

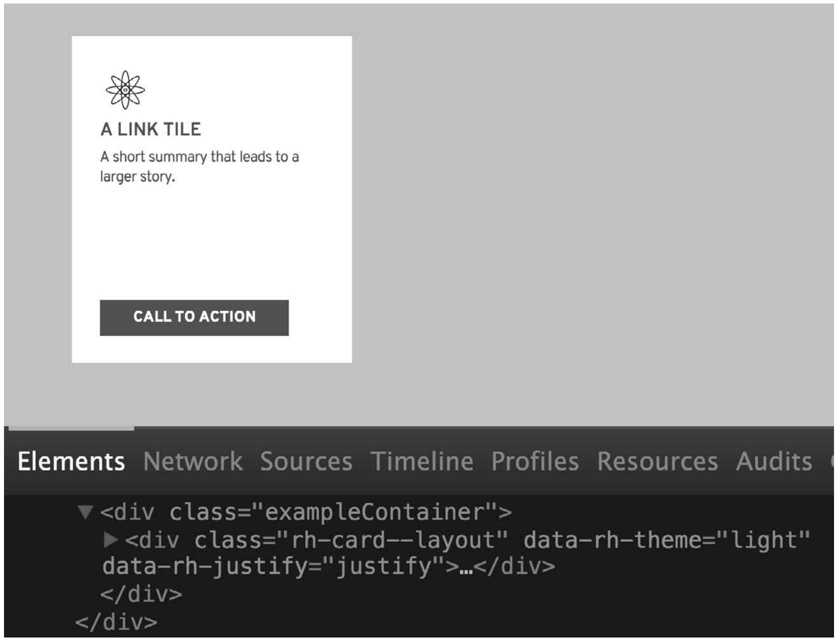


图 7-8：这个浅色主题的卡片有一个黑色的图标

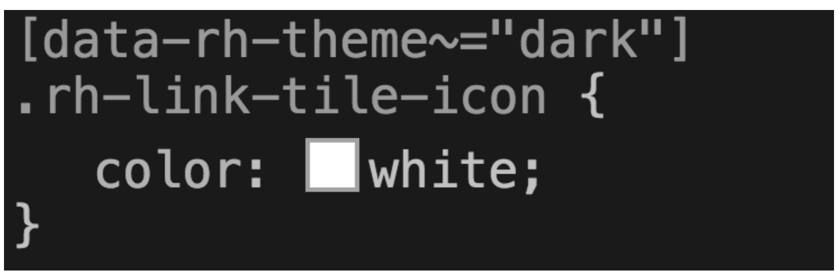


图 7-9：在深色主题下图标配合显示白色



图 7-10：在浅色主题下图标配合显示黑色

7.8　语义化的网格

随着组件的基础变得坚实，我们需要做的下一件事是弄清楚如何把它们合并成多种不一样的布局。过去，当模块数量比较少时，我们会给每个版块写样式，包括版块的布局。如果要做一个充满商标的版块，我们会给这个版块一个 CSS 类名称，然后明确地给这个版块应用一个布局。这个方案的问题是，除了整个版块其他的都不可复用。所以如果有别的版块用到了相同的布局但不同的内容，我们就不得不创建一个全新的版块以及重写布局样式。

所以，现在我们把商标墙拆分成图片集、按钮和标题，希望模块布局任何时候都能复用。这意味着要把布局放回 DOM，这正是我们舍弃 bootstrap 网格布局时强烈抗拒的事情。

幸运的是，bootstrap 网格布局容器、行容器和列容器，需要应用到 `.logo-wall` 版块的特定布局，以及其他任何需要布局的新的内容组合。

我们的解决方案是，创造一组可以通过数据属性的值来应用的常用网格布局方式（见图 7-11）。只要在父节点的标签上设置布局属性，所有的子标签都会应用相应的网格布局类型。这种方法让我们不仅可以将布局和组件分离开，同时还可以在标记上设置网格布局。

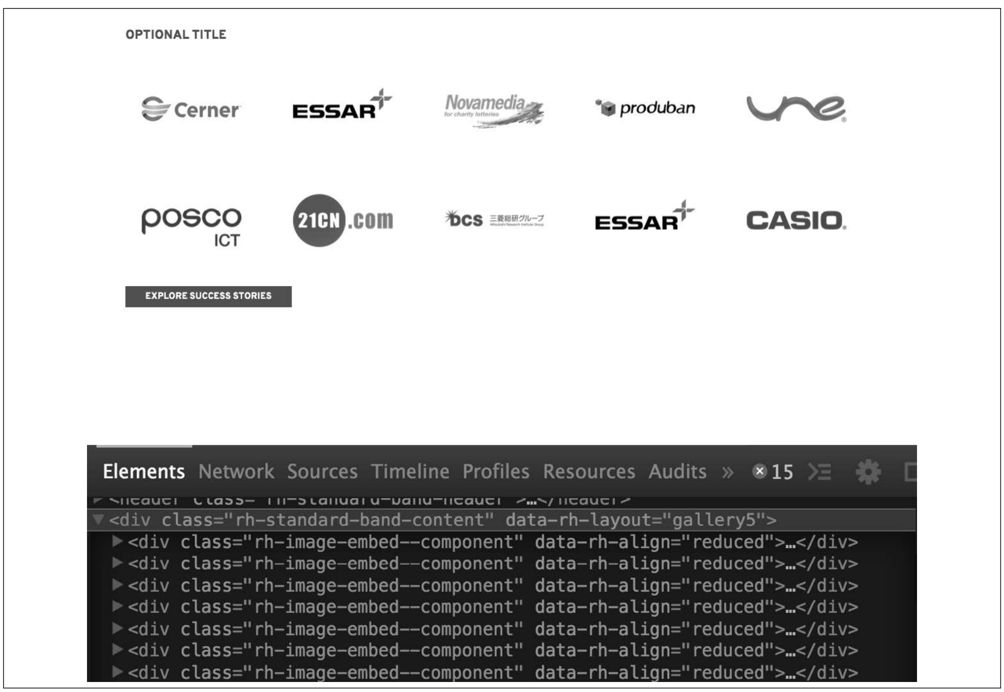


图 7-11：一个单独的数据属性让我们可以在一大组图片上设置布局

前面的例子里，你可以看到我们给版块内容区域（版块里的一个主要内容区域）应用了一个 `data-rh-layout` 属性。这个属性再一次体现了数据属性在组织大量内容 / 数据的情况下

是多么有优势。只用了一个属性，我们就能传递多个不一样的布局，来设置子组件的宽度和外边距。

在这个例子里，传入了 `gallery5` 属性以后，CSS 样式给所有的子节点应用了大致 20% 的宽度。通过在父元素的标签上标设置好所有的网格信息，我们就可以将不同的组件内容组合起来放在一个内容区域中，而不需要另外添加 CSS 类或者把组件放在额外的行容器、列容器里。因此，当我们想把商标墙从 5 列改成 4 列，唯一需要做的事情就是在父标签上变更一个属性（见图 7-12）。



图 7-12：改变了单一的数据属性，我们就可以改变这些商标的布局

现在，我们已经可以通过变更一个属性来修改整个版块的布局了。我们再也不必不断地写新的 CSS（实现了模块化）了，也不必再给内容添加 CSS 类（实现了语义化）了。一箭双雕！现在我们有一个图片集清单和 12 列（6 6、3 9、4 4 4 等）的普通布局组合，它们涵盖了我们需要创建的 99% 的布局。如果需要一些更独特的布局，我们可以创建一个自定义的网格布局，写好说明，让任何人都可以应用到他们的内容上。

如今，我们已经有了一组模块化的和可定制的容器，这组容器可以容纳我们的组件，同时给组件应用布局。之后我们就可以建立商标墙、专题活动、博客广告以及网站所需要的其他所有版块了。

第三部分

流程核心

流程核心的意义在于清晰地定义前端代码从开发人员的脑海到用户的浏览器所需要经历的各个步骤。流程包含了开发过程的各个环节，从合理的想法到可行的设计，到有效的提交，再到最终的部署。

如果你从事前端开发的时间足够长，你可能注意到，在过去的几年内，我们的工作流已经发生了巨大的变化。我第一次做 Web 开发是通过两个月的邮件交流来理解客户的需求，然后通过 FTP 登录他们的服务器，对网站代码做必要的修改的。

现在看来，这种更新网站代码的方式非常糟糕。如果我误解了邮件的内容，改错了代码，会发生什么呢？如果我不小心删掉了一大片 CSS 代码，又不小心破坏了网站上的其他页面呢？如果我改掉一个 JavaScript 的 bug，但又引发另外两个 bug 呢？这些问题十分棘手，而这也是永远都不要用 FTP 直接修改代码的重要原因。如果你没有经常备份，现在需要修复一个损坏的网站，而且还有一堆任务等着你去做，那么结果会怎样呢？

幸运的是，我们大多数人已经从这些错误中吸取了教训，现在遵循着更先进的方法。跟之前通过 FTP 的方式来修改邮件中提到的各种改动不同，现在我们采取的方法如下。

(1) 使用事件跟踪和用户故事来正确地跟踪工作流和标记那些完成了的任务。
(2) 搭建开发环境来测试代码。
(3) 构建部署流程，用于编译、验证和测试代码。
(4) 在任何代码被采纳之前，都要获取需求方的反馈。
(5) 把提交的代码推送到中心代码仓库。

(6) 采用这样一个部署系统：可以无缝地添加一些新代码到生产环境；在需要的时候，还可以回滚这些代码来恢复系统原来的功能。

准备好实践这些方法了吗

作为前端架构师，你经常需要制定，或至少能够掌控以上所列的每一项内容。流程中的任何一个环节出现问题，都会迅速演变为开发人员的痛苦，或者导致网站无法持续满足用户要求，甚至崩溃。

正如前面的章节所说，前端架构师的用户是开发人员。我们所选择的工具、编写的代码、创建的流程，都是为了让开发人员能够构建出最高效的、不出错的、可扩展的和可持续优化的系统。

在这一部分，我们会从一个更高的视角看看如何更好地武装新的开发人员，让他们更快地上手。我们会深入了解他们是如何把需求转化为实际代码的，以及代码产生的流程。然后我们会讨论这些流程的核心——任务处理器的角色，以及它们如何帮助我们用更少的时间创造更好的代码。

第 8 章

工作流

流程的核心是工作流。工作流指的是把想法变成现实的过程，或者从产品的角度来看，就是解决 bug、需求迭代的一系列流程和方法。然而，前端工作流不是凭空产生的，关键是要结合整个团队的实际情况，在实际的开发情景中不断细化，从而制定出适合自己的工作流。

8.1　过去的开发工作流

对此，我一定抱怨过无数次了，但仍有必要再次说明。不要使用图 8-1 那样的方式作为你的开发工作流。单纯地对着 PSD 文件编写一堆标记的时代已经过去、结束甚至死亡了，而且最好埋葬在一个再也不会有人发现的地方。前端开发不是为了在一堆混乱的 HTML 代码上添加漂亮的样式，我们也不是仅仅为了把页面弄得更漂亮。

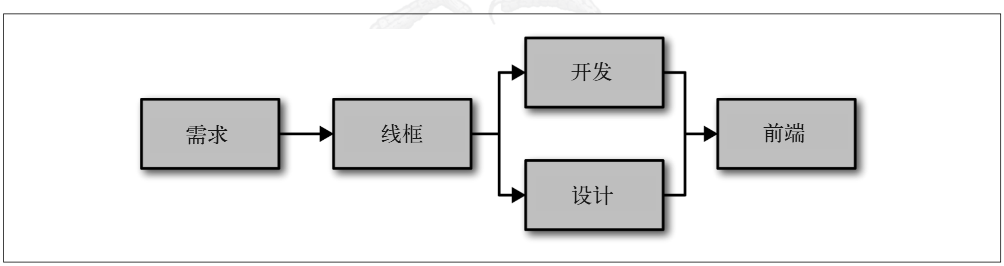


图 8-1：过去的开发工作流

8.2 现代的开发工作流

过去的工作流是根据角色逐级交付的，由产品负责人交给体验设计师，投入具体的开发之后再由视觉设计师交给前端工程师。然而，现代的工作流则完全相反。为了创建高性能、响应式、易用性高的网站，我们必须把以前的工作流抛诸脑后。这样一来，我们就得到了类似图 8-2 的工作流。

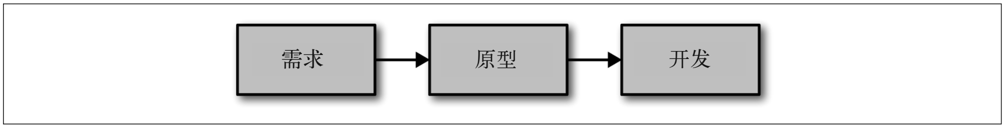


图 8-2：现代的开发工作流

8.2.1 需求

工作流一般是从收集需求开始的，因为只有这样我们才能够定义出项目内容和衡量项目成败的标准。新旧工作流的关键区别在于需求所面向的人群变了，其中包括交互设计、视觉设计、后端开发，以及前端开发人员。这样一个来自交叉领域的团队，意味着我们将会注重创建一个完整的终端解决方案，而不仅仅是一个大概的线框图或者一个静态的页面。

通过让这四个领域的人员共同参与需求收集的过程，我们可以更早地发现需求中存在的问题和不足。因为我们经常等到开发工作已经完成或者项目的交付日期马上到来的时候，才发现代码次序错乱、影响性能的设计或者糟糕的用户体验等问题。

8.2.2 原型设计

跟以往每个环节直接交付一个成品不同，新的工作流注重在用户交互模型、视觉设计和前端解决方案中持续迭代。

原型设计提供了一个讨论和反馈的公共空间，它把丰满的想法实现在桌面和手机浏览器中。在原型中，想法可以成型、摒弃、重拾、打磨。一旦开发人员和产品负责人对这个原型满意，那么它就可以被采纳了。

直到原型（制作成本低而修改方便）被确认采纳，我们才进入开发环节（制作成本高而不灵活）。

8.2.3 程序开发

在这个环节，我们的需求可以实现得更加容易和顺利。不仅是因为我们从一个经过充分验证的设计开始，而且此时我们已经拥有了实现这个设计所需要的全部标记。开发人员的工作就是收集和处理来自数据库的数据，然后把它们放置到对应的标记上。开发人员几乎不

需要给标记添加额外的类，或者去掉一个容器 div，或者调整代码的顺序，因为所有的迭代和测试工作都在原型阶段完成了。

在你的项目中，如果原型和网站共用一套 CSS 和 JavaScript，那么开发人员应该完成一个功能完整、样式齐全、交互良好、响应及时、通过检查和达到标准的产品。原型设计不仅有助于开发团队达到这一目标，对于测试团队也是一个好消息。以往测试人员要参照传统的、甚至已经过时的需求文档，来判断开发工作是否正确，如今他们可以根据原型设计这个黄金准则来一步步地检验开发的内容。

8.3 前端工作流

从宏观角度切换到微观角度，我们来了解一个良好的原型设计流程的重要性，以保证你团队中的前端工程师都已经做好迈向成功的准备。这里的很多内容都是跨界的，但是它们对前端工程师的开发效率和工作满意度有着重要的影响。

现在，我们不能假设所有的工程师都有相关的经验，因此前端工作流应该在新人入职时就开始运作。这样一来，理解一个新人工作时需要的所有步骤很重要，包括第一次坐下来面对代码，用新笔记本电脑写下他们人生中第一行有效的代码。

8.3.1 必要的工具

所有新来的工程师的首要任务都是安装必要的软件和搭建代码运行的环境。这通常包括安装和配置你最喜欢的代码编辑器、安装一些 Adobe 公司的软件，以及下载你常用的浏览器。在这些软件跑起来之后，往往需要几个步骤去安装 Git 和配置服务器准入。最后，新人还需要整理各种各样的网络服务和密码。

没错，这看起来有点普通，但是这个流程越流畅，工程师们就能越快地进入到实际的编码工作中。

8.3.2 本地部署

跟版本控制器打交道，往往是工程师上班时要做的第一件事，也是他下班前做的最后一件事。这是一个授权他们访问网站代码和让他们发布新代码的工具。这个时候，他们可能需要克隆你的代码到本地环境并且使其在他们的机器上运作起来。这个步骤可以很简单：

```
git clone <repo> && cd <repo-name> && make website
```

或者，这可能是个很长的流程，包括拉取多个代码库，安装一个本地数据库，配置各种各样的服务器设置，甚至修改计算机网络和设置 VPN。不管你的流程是什么，一定要确保在 README.md 文件中说清楚，以及给出可以联系的人或者资源，以防用户对流程中的某些步骤存在疑问。

千万不要低估这个流程所需花费的时间！我曾经参与过一些只需要几分钟就能配置起来的项目，我也听过一些糟糕的故事：一位新的工程师花了几个星期才使网站在本地环境跑起来。

所以，现在我们的工程师拥有了一个代码编辑器、一个浏览器和一个本地可编辑的网站。那么，在他们开始工作之前，还需要什么呢？不妨给他们一些事情做？我们将在下一节介绍。

8.3.3　编写用户故事

不管你叫它们任务、标签、故事还是作业，我们都需要通过沟通把人的想法和愿望提炼成一个精准的、可操作的和可检验的要求。因为没有人会读心术，所以我们需要编写用户故事，详细地描述问题的缘由、建议的解决方案和必须满足的需求点。

把所有的事情分解成小而简单的需求，这种方式在前端开发中也是适用的。现在，既然从简单的开发页面进入到了构建设计系统的级别，我们就需要确保分解工作任务的方式能够体现新的方法论思想。这意味着我们不需要再写类似“更新‘关于我们’页面”这样的用户故事。这一类需求通常包含一系列的排版和布局上的变更，还有可能包含类似“把用户行动召唤（call to action，CTA）按钮内边距扩大一倍”这样的要求。以下解释了为什么说这样的用户故事非常糟糕。

- 我们是否被问及，是改变所有 CTA 按钮的内边距，还是只改变“关于我们”页面的那一个？
- 如果我们不是要更新所有的 CTA 按钮，那么除了“关于我们”之外，是否还有其他页面需要采用这个大号的按钮？
- 提出这个任务的人，是否有权做全站范围的改动？或者，假如我们改变了全部的 CTA 按钮，但是正在为主页编写需求的人却不希望使用这个新按钮呢？
- 如果一个人正在更新“关于我们”页面上的 CTA 按钮，但另外一个人又被指派去改变“联系我们”页面上的 CTA 按钮，会发生什么呢？
- 为什么对全局 CTA 按钮的一个简单调整，会跟一堆的局部和全局的变动糅合在一起？

比起那些对单页面进行多处修改的任务，我们更应该把前端开发任务的重心放在对系统所做的改变上。与其编写一个涉及一堆修改的大任务，不如来一个这样的任务：“创建一个新的拥有 16px 内边距的 CTA 按钮，用于代替所有内部网页中那些 8px 内边距的按钮。”

很明显，这样的任务不是让我们去改变原有的 CTA 按钮，而是创建一个新的组件。我们都知道，这样的改变是可选的，只有我们回到“关于我们”页面（另外一个任务），更新 HTML 去使用新的组件时，网站的内容才会发生变化。

通过关注组件而非单页面的内容，我们可以保证优先考虑的是设计系统，以及改动对系统产生的影响。这样的方式创建了一个更具弹性的系统，有助于避免多个页面之间的冲突。

8.4 开发

现在我们的任务是更新系统而不仅仅是更新页面，我们可以设计一个对系统影响最小的改动来实现这个需求。我们已经有一个拥有 8px 内边距的典型的按钮，现在我们需要新建一个拥有 16px 内边距的组件。先来看看要改动的代码：

```
.cta {
  background: blue;
  color: white;
  padding: 8px;
}
```

在工程师编写新的样式之前，我们应该让他们遵循一个普遍而公认的方法，那就是创建功能分支。功能分支是一个从开发主干分离出来的 Git 分支，你可以在上面开发和提交代码，并最终把功能特性合并到主干中。这个方法的好处是，你不用向主干提交不完整的代码。另外，在合并回主干之前，其他人可以检查并测试你在功能分支上提交的代码。

在新的功能分支创建以后，是时候让工程师编写 CSS 了。现在只有一个问题：工程师写这个 CSS 可以有很多种方法。

他们可以用一个全新的按钮来代替之前的那个：

```
.cta-large {
  background: blue;
  color: white;
  padding: 16px;
}
```

如果正在使用 Sass，那么他们可能用混入来处理这个任务，然后重构这两个按钮：

```
@mixin button($size) {
  background: blue;
  color: white;
  padding: $size;

}

.cta {
 @include button(8px);
}

.cta-large {
 @include button(8px);
}
```

也可以选择添加一个用于重置样式的类，这样我们的按钮就会具有 `.cta` 和 `.cta-large` 两个类：

```
.cta-large {
  padding: 16px;
}

/* HTML示例:<a class="cta cta-large"> */
```

或者可以使用数据属性来代替类，写法如下：

```
.cta[data-size="large"] {
 padding: 16px;
}

/* HTML示例：<a class="cta" data-size="large"> */
```

很明显，就像修改一下按钮这么简单的事情，也有无数种不同的方法。虽然上面这四种方法的效果是一样的，但对于创建和维护的系统而言，影响却大相径庭。

这就是为什么我们要在第二部分花那么多的时间说明我们书写选择器、CSS 和标签元素的方式。有了描述代码规范的文档，工程师更容易编写出正确的代码，并提交到它们的功能分支上。

这时，可以检查新开发的功能分支了，并且工程师可以创建一个合并请求（或者拉取请求），这等于说："我这里有一些我认为完整的代码，请你检查一下，然后合并到主干好吗？"这会比直接合并代码到主干费时一些，但是好处是在这些代码被采纳之前，拉取请求提供了一个代码审查和意见反馈的机会。

有时候，审查的过程中会发现代码中的一些重大缺陷，或者觉得这些代码不符合系统的设计原则。有时候这个审查只是提供一个机会给出更好的建议，比如更佳的 CSS 编写方式、更优的 Sass 函数、更清晰的文档说明。就算最后的合并请求只是得到了一个大赞，多一个人检查系统的改动，也有助于保证提交的代码是最佳的。

现在，工程师创建的代码已经被合并到主干，是时候把这些代码部署到生产环境中了。

8.5 发布

我们的工程师做到了！他们在代码库中做了一些改动来解决 CTA 按钮的问题，经过代码审查和测试，我们已经准备好把这些改动发布给用户。这一次，假设我们采用的是 Sass 的混入方式去创建一个 `.cta` 按钮和一个 `.cta-large` 按钮。那么，Sass 文件的改动被提交到了主干，但是这并不能保证用户在访问网站时能够使用到这些新的 CSS。因此在你打开 FTP 客户端之前，让我们看一下不同发布方式的利弊。

8.6 提交编译后的资源

在版本控制器中，最好的方法是只提交少量必要的代码。这意思是说，在 Git 中忽略那些

临时文件，或者那些需要下载用来编译代码的资源文件（我们会在下一章中详细介绍）。这样做的理由是网站并不需要那些临时文件和 Node 模块。这些文件不但明显地增加了项目文件的体积，而且还会不断引起代码冲突，因为你本地的临时文件可能跟版本中那些对应的最新文件不一致。

不幸的是，有一种文件类型正好介于“网站功能所需”和“臃肿且引起代码冲突”之间：编译后的资源文件。当编译 Sass、CoffeeScript、LESS 时，通过后置处理器添加浏览器厂商前缀时，合并 JavaScript 文件时，创建图标字体或者压缩图片时，我们都在创建编译后的文件。我们应当怎么处理这些文件呢？如果我们提交它们会怎么样呢？能否实现即使不提交它们，网站仍然能正常运行呢？

很多场景下，你会发现不得不提交这些资源文件。这不是理想的方法，除非你想抛弃预处理器并且手写 CSS 代码，否则你还是得把它们添加到源码中。这个方法的好处是，你的源码中永远保存着渲染网站所需的所有文件。如果你的服务器是直接读取 GitHub 中主干的代码，那么你将拥有网站所需要的一切。另外一个好处是，如果那些不懂技术的用户复制一份你的网站代码，他们可以直接访问网站，而不需要经过漫长的搭建编译工具的过程。

当然，享受这个小小的好处的同时，我们也要面对一系列的问题，其中最大的问题就是合并代码时的冲突。如果你曾经在一个拥有编译后资源的项目中执行过 Git rebase 指令，就会明白我想表达什么。就算不是 rebase 指令，只要你合并两个具有编译之后代码的分支，也很有可能需要处理那些编译后的资源所带来的冲突。现在，解决这些冲突的方法很简单（重新编译一切并且提交），但是这也意味着，你永远不会有合并请求，可以让其他前端工程师一起来解决最终 CSS 中的代码冲突。

如果你希望避免这些问题，并且在你的代码库中不保存编译后的资源文件，现在你有了一些选择。接下来让我们看一看。

8.7 持续集成的服务器

使用类似 Jenkins 或 Travis CI 的服务带来的好处远远超出了本书的范围，不过它们的众多用法之一是在代码发布到服务器之前，先对代码进行一些处理。这意味着我们可以在 Git 上忽略那些编译后的资源，因为 CI 服务器会在检测代码后执行我们的编译任务，然后再把代码发布到服务器。

这种方法不仅可以保持代码库的整洁，也有助于减少意料之外的代码回归。如果我们把编译后的资源文件提交到代码库，那么当我的功能分支合并到主干时，主干上的那些资源文件肯定就是我的机器上编译的结果。一旦有其他工程师的功能分支合并到主干，那么我的功能特性就会被那个工程师电脑上编译的结果所覆盖。

当然，其他工程师也是基于我最新的 Sass 代码和 JavaScript 文件来编译的，但是往往由于操作系统、系统设置或者安装软件的不同，最终编译出来的代码也会有所不同。另一方面，如果我们使用 CI 编译的代码来测试功能分支，同时也使用 CI 编译的代码部署到线上，那么任何时候我们都能保证这些编译后的代码是完全一样的。

8.7.1 标签分支

Git 的强大功能之一就是创建标签分支。标签是项目历史中的一个快照，基于某一次提交后的代码。不管是发布到线上服务器，还是发布渠道（我们后面会介绍），标签是发布代码的一种便捷方式。标签的一个好处是，它可以基于任何分支，而不仅仅是线上分支。

8.7.2 究竟为什么要这么做

发布软件的一个方法是为软件的每个版本创建一个发布分支。如果 1.1 版本准备发布，你就从主干创建一个名叫 v1.1-dev 的分支，然后在这个分支上继续开发，直至版本发布就绪。通常来说，如果 v1.1-dev 这个分支上有一些 bug，你会在分支上解决，然后选择是否要把这些修复问题的代码合并到主干。有时候为了发布 1.1 版本，我们需要那些修复问题的代码，但这些代码跟最终合并到主干的不一定一样。你可能只是临时修复一下，把问题记下来，以后有时间再用正确的方法解决。

有效利用标签分支的另一个方法是只在那些标签分支上提交你编译后的资源。记住，标签永远指的是某一次提交，即使创建那次提交的分支被删除之后，标签仍然存在。而且，这一次提交也没有说一定要合并到主干上。在第 10 章中，我会介绍我们在 Red Hat 项目中使用的流程。现在，在结束这一章之前，让我们看一下如何利用发布渠道把我们的代码发布到多个服务器而非单一服务器。

8.8 发布渠道

如果你所有的代码都在一个代码库中，那么你可能不用太多地关注发布渠道。但如果你是在为几十个甚至几百个网站编写主题、类库或者模块，除了简单的版本控制器之外，你可能已经尝试过不同的发布渠道。这些渠道比你想象的要多，而且你经常有不止一种选择。以下是一个简短的、可能不完整的列表，至少能让你有个了解：

- NPM（Node Package Manager）
- Bower
- Ruby Gems
- Composer
- Yeoman Generators
- Drupal Contrib Modules 和 Drush

- WordPress 插件
- Homebrew
- Sublime Text Package Control
- RPM
- PEAR

不管写的是哪种类型的代码（PHP、Node、Ruby、CMD 模块、Mac 软件、Unix 软件、VIM 插件、IDE 插件），你都会发现已经有人为它编写了用于发布代码的包管理器。考虑到 Bower 其实是个 Node 模块，所以说甚至还有包管理器来管理你的包管理器。使用包管理器来发布你的代码有很多好处，这些好处比我们需要的还多，因此包管理器当然比其他的工具更好。

软件包管理器的好处如下。

发布不同的版本

这代表使用你代码的用户不会自动升级到最新版，他们可以通过评估新代码来决定是否需要升级。

用户很容易知道什么时候有新版本可用

大多数的包管理器都拥有内置的通知机制和内部的升级系统，因此我们很容易获取新的模块和代码。

只发布那些用户所需的文件

你往往有一个很大的系统来编译你的代码，而使用包管理器可以让你只发布那些用户需要的文件。

从私有代码库中发布代码

有时候 Git 代码库的权限不能开放给所有人。即使最终产品面对的是普通用户，私有版本的控制器和防火墙也会将源码加以严密保护。包管理器允许你把那些发布的代码放置到公共的空间。

花些时间看看你的项目的一些选择。你不会永远只停留在一个单一的系统上（很多项目都是部署在 Bower 和 NPM 上），然而不管你决定做什么系统，都会因为用户使用你的代码来构建编译系统而感到自豪。因此，接下来我们讨论一下用来创建这些共享代码的系统，以及以其他人编写的代码为基础的系统。

第 9 章

任务处理器

对很多前端工程师来说，Sass 是接触到的第一个编译系统，或者说是任务处理器。在这之前，我们的工作方式通常是修改 CSS 文件然后刷新浏览器。Sass 打开了自动化处理的大门。有些人可能会认为我们只是在工作流中多增加了一条指令，但对于那些第一次发现 Sass 的人来说，这个小小的任务是非常值得一试的。

Compass 让我们不仅发现 CSS3 的混入可以节省很多查询 CSS3 样式生成器（http://css3generator.com/）的时间，还认识到了项目中配置文件的作用。有了 Compass，我们可以创建一个 config.rb 文件，用来描述重要文件夹的路径、指定开发环境与线上环境的设置，并绑定重要的调试工具。这个配置文件可以保留并分享给团队的其他人，让新人更容易上手。

随着 Sass 社区的发展，越来越多的 Compass 插件涌现出来，从先进的颜色函数到栅格系统、布局比例工具、媒体查询管理，等等。这些工具都作为 gem 组件发布，可以直接导入任何一个使用 Compass 的项目中。

Compass 还可以让模块开发者使用 Ruby 的功能，比如访问本地文件、执行 Ruby 函数、编译 Ruby 模板和把数据传递回 Sass 文件。这允许用户在模块中进行复杂的数学运算，而这是单独使用 Sass 所无法实现的。Compass 还具备这样的功能：用不同文件夹中的 PNG 图片生成雪碧图，或获取图片文件的宽度和高度。

如今只用一条 `compass compile` 指令，开发人员就可以把所有的 Sass 文件编译到目标 CSS 所在的目录，并且图片和字体都分别指向各自的路径。Sass 文件可以使用流行的框架来编译，如 Susy grids、Modular Scale、Breakpoint 和 Color Schemer。Compass 在创建若干个自

定义的雪碧图，批量获取图片的尺寸，并在 CSS 中设置它们的高度和宽度时，就会完成上述的工作。

Compass 不仅是我们对编译系统的第一次接触，也是很多人不愿使用最近很火的新型 Node.js 任务处理器的首要原因。简单了解 Grunt 和 Gulp 这类工具之后，很多人（包括我）的第一反应都是："噢，这跟 Compass 差不多，只是更复杂些而已。"对于一个完全使用 Compass 来构建编译流程的开发人员来说，配置 Grunt 来编译 Sass 像是一个完全不必要的步骤。然而，一旦配置好 Gulp 或 Grunt 来编译 Sass，我们就开始关注其他可做的事情，并且很快意识到我们的编译流程与之前相比还是有所不同的。

9.1 在任务处理器中完成一切

使用基于 Node.js 的任务处理器，在我的工作流中有一点多米诺骨牌效应。作为一名 Grunt 用户，我拥有 13 000 多个可用模块，而且这些模块为我的工作流提供了各种各样的功能，而不单单是 Sass 和 CSS。下面是我使用任务处理器实现的一些功能：

- 安装需要的 Ruby 库和 Bower 安装包
- 清理临时文件夹
- 创建软连接
- 编译 Sass
- 合并 JavaScript
- 加载第三方 JavaScript 库
- 把 SVG 文件编译成图标字体
- 对图片进行处理，减少文件体积，裁剪成各种尺寸
- 同步文件到远程服务器
- 创建 Git 标签
- 运行可视化的回归测试
- 生成代码的样式文档
- 自动生成浏览器厂商的前缀
- 编译组件库
- 优化我的 Sass、CSS、JavaScript、JSON，等等
- 基于 JSON 模式来验证数据
- 启动 Node 和 PHP 服务器
- 监听文件改动来刷新浏览器

任务处理器使前端架构师能够创建网站的蓝图。四个核心都被打包成自动化流程，通过自动化技术，我们不仅规范了流程，同时还对它进行了优化。代码检查工具能够帮助我们规范和提高代码标准。测试组件在本地或者持续集成工具（比如 Jenkins 或 TravisCI）中运

行。只需点一下鼠标，就可以持续编译出新的代码并发布到线上。此外，文档还可以自动读取代码中的注释、模板文件和描述大纲。

更深的探索

刚接触前端架构时，我还没听说过 Grunt，那时对创建 Compass 模块非常感兴趣。我从编写可以支持新项目的小型 gem 包开始，思考怎样利用 gem 包把一个基础样式同时发布到多个网站上。虽然这两个例子都是有效的，但它们反映出，如果我对 Ruby 了解不多，那么能做的事情将会非常有限。再者，学习了一些基本的 Ruby 教程之后，我意识到除了 Ruby on Rails 应用相关的知识，市面上并没有太多学习 Ruby 编程的资源，更不用说使用 Ruby 编写 Compass 模块的了。

对于 Grunt 而言，我的优势之一是已经掌握了编写模块所使用的语言：JavaScript。在使用一些模块几个月之后，我发现它们已经能够满足我 95% 的需求。虽然我现在还不是 Node.js 开发者，但在某些场景下还是可以给一个模块添加特性或者修复 bug，然后把它发布出去。有一次，我想在流程中使用 Grunt-PhantomCSS 模块，但是发现必须重写项目的大部分代码来适配它。因此，我复制了这个项目，做了一些必要的改动并提交到 GitHub 上，这样其他人就可以在他们的项目中使用它了。我成为了一名模块编写者，而不仅仅是模块使用者。

9.2 在项目中使用任务处理器

我第一次接触 Grunt 是在一个大型的现成的代码库中，它有众多的模块和抽象的代码组织方式，因此对新手来说不容易理解。然而，经过一段时间的学习，最终我能够修改部分代码，甚至添加一些新的功能进去。不过，我是一个需要从头到尾理解代码之后才能接受和投入其中的人。

然后，我开始分解项目，移除一切能够移除的东西，直至剩下可运行的最小基本框架和基础配置。这时，我能理解项目中的每一行代码，并且可以用我自己的方式在项目中实现更复杂的功能。下面是一段用于编译 Sass 的最简单的 Grunt 代码：

```
module.exports = function(grunt) {
 grunt.loadNpmTasks('grunt-sass');
 grunt.initConfig({
   sass: {
     options: {
     },
     dist: {
       src: 'sass/style.scss',
       dest: 'css/style.css',
     }
   }
```

```
  });
  grunt.registerTask('default', [
    'sass'
    ]);
};
```

上面这段代码的作用是把你的 style.scss 程序代码编译成 style.css，可以分为如下三个部分。

加载任务

Grunt 需要知道加载哪个 Node 任务到编译流程中。如果你直接加载 package.json 中的所有的包，那么 `load-grunt-tasks` 模块（https://github.com/sindresorhus/load-grunt-tasks）将会减轻你的很多负担。

设定配置

每一个加载的任务都有一系列可以配置的选项。在这个例子中，我们只是配置了 Sass 任务的源地址和目标地址，实际上还可以打开或关闭 sourcemaps、设置额外的包含路径，或者改变最终输出的 CSS 格式。每个任务都可以有不同的写法或者目标。你可以设定一个打开 sourcemaps 和输出完整代码的配置用于开发环境，然后设定一个关闭 sourcemaps 和输出压缩代码的配置用于线上环境。

注册自定义任务

在这个环节，我们可以把各个独立的任务整合到一个自定义的父级任务中。`default` 任务是运行 `grunt` 指令所执行的默认任务。你可以在 `default` 任务中添加更多的任务，比如 Sass Lint，也可以新建其他自定义的父级任务，比如 `test` 或 `deploy`。

现在用 GulpJS 实现同样的功能：

```
var gulp = require('gulp');
var sass = require('gulp-sass');
gulp.task('default', function () {
   gulp.src('./*.scss')
       .pipe(sass())
       .pipe(gulp.dest('./css'));
});
```

Gulp 定义任务的方式跟 Grunt 有所不同。从脚本中你就能发现它们在代码风格上大相径庭。Grunt 基于配置使用对象的符号，在一个地方定义任务，然后在另一个地方执行任务；而 Gulp 把任务和配置连在一起，使用管道的方式，把代码从一个操作传递到下一个操作。接下来分析一下这段代码。

加载模块

与 Grunt 使用 `loadNpmTasks` 不同，Gulp 使用传统的基于 Node 的 `require()` 语法。因此一开始我们就把 `gulp` 和 `gulp-sass` 加载进来，并保存到变量中供后续使用。

创建自定义任务

这里再次使用了 `default` 任务，表明运行 `gulp` 时会发生什么。不过，我们并不是使用 `grunt.registerTask` 来列出所有想要执行的预定义任务，而是在函数内部来构建完整的处理流程。

Gulp的优点

Gulp 方式倾向于使用小而并行的函数，这些函数往往先是收集资源，然后通过几个管道处理，最终把结果输出到目标环境。它主要有两个优点：一、并行执行的方式意味着在处理 Sass 的时候，不会阻塞其他任务的运行；二、管道的方式让 Gulp 可以连贯地对一个资源进行多个操作，而不是像 Grunt 那样，需要把输出的 CSS 保存到一个临时目录，然后再对这些临时文件进行第二次处理。

9.3 有明显的优胜者吗

那么，哪种方式更好呢？这取决于你的需求。Grunt 是目前的主流方式，优势是拥有大量的现成模块。另一方面，Gulp 在大型项目中的执行速度更快一些，它使用管道的方式来处理代码，可以让你的项目变得更加精巧。

尽管我想尝试一下 Gulp，看看它在我的项目中的效果，但几年来我一直愉快地用着 Grunt。总体来说，两者都是非常优秀的工具，如果它们都具备你需要使用的模块，那么最终影响你决定的可能就是代码风格了。

最后，任务处理器只是一个工具。前端架构师的职责在于创建高效且抗差错的工作流。因此，如果你的工具能帮助开发人员快速地运作起来，让他们在健壮的环境中编写高质量的代码，然后把代码部署到测试、预发布和正式发布环境，那么不管你选择的是什么框架，你都是称职的前端架构师。

第 10 章

Red Hat流程

在 Red Hat 公司，前端团队有着在后端开发之前就进行多次迭代的绝佳优势。我们先对想要创建或更新的功能设定长期的目标，然后一拿到经过确认的原型，就交给开发团队去实现。

在过去的项目中，我们也许说过：“好了，这就是我们希望输出的标记格式，请尽可能保持一致。”我们会使用一组 Mustache 或者 Twig 模板来创建标记，然后告诉开发团队去创建能够输出同样标记的 PHP、Ruby、Angular、React 或 Ember 模板。这种方式的问题在于，有一半的开发时间都浪费在让后端输出的标记跟我们的原型保持一致上了。因为我们总是在解决“CMS 中输出的标记与我们的原型不同”的问题。

即使我们真的做到了让那些标记跟原型保持一致，往往也会担心原型和 CMS 代码不同步的问题。有可能其中一边发生了变化，另一边却没有，最终慢慢导致我们的原型不被信任。我们不能确定原型与正在开发的产品是否一致，对功能的迭代又是直接在 CMS 代码上做改动。这个问题的原因在于，CMS 和原型开发工具使用的是不同的 HTML 模板，即使它们共享了 CSS 和 JavaScript 文件。在对 Red Hat 网站的主题进行重构时，我们有机会彻底解决这个“最后一英里”（last mile）的问题。

10.1　征服最后一英里

我们的问题是，Drupal 有一个非常固执的渲染引擎，因为标记必须通过 Drupal 才能渲染，所以如果不运行 Drupal，就很难制作标记的原型。因此，我们决定颠覆一下 Drupal 的渲染途径，并加入自己的解决方案。我们想要一个可以在原型工具、Drupal、WordPress，以及

其他任何希望采用我们的设计系统的平台上都能使用的解决方案。我们从 Twig 模板语言入手，因为它成熟且稳定，还拥有 PHP 和 Node 编译引擎。

既然我们已经有了一个通用的渲染引擎，它允许在原型工具中使用跟实际开发中一致的模板文件，那么就再也不用担心系统之间转换模板的问题了。一旦在样式文档中创建了一个样例，我们就可以把用于那个样例的数据集记录下来，交给后端开发人员来实现。在某种程度上，我们创建了一个设计系统的 API，或者一个样式服务，它为用户提供了一个简单、清晰、跨平台的接口，用于接收数据输入和输出相同的 HTML。

一系列标准的产出

数据和模板的关系很快成为我们的设计系统中非常重要的部分。开发人员开始要求提供模板变量的列表、可变的内容类型，以及它们当中哪些是必选项。这些要求很重要，我们也希望给开发人员提供尽可能完整的信息。于是，就有了和所有的组件和布局一同提供的一系列标准产出。

一系列标准的组件产出如下。

JSON模式
: 该模式定义了组件的各种属性（变量）、内容类型，以及其中哪些是必选项。

模板文件
: Twig 文件接收由 JSON 模式定义的一组数据。可选数据被放在 `if` 语句之后，而数组数据则会被遍历。

Sass partial
: 所有的组件样式都来自单一的 Sass partial 文件[1]。它会被编译到主样式 style.css 中，或者作为一个独立的 CSS 文件被加载进来。

可视化的回归测试
: 这个文件描述了每一次运行回归测试时，需要测试的所有的浏览器宽度和组件状态。在加入任何新的代码前，一定要通过这些测试。

测试数据
: 这个文件允许构造测试专用的数据来达到完整的测试覆盖率，以及测试一些边界情况。

文档
: 文档文件是为文档工具 Hologram（https://github.com/trulia/hologram）提供数据的

注 1：这是一种通用的命名惯例，通常这个文件不会被它自己编译，但在某些依赖关系的情况下，也可能被自己编译。——译者注

markdown 文件，它提供组件的描述并列明其功能。此外，文档页还提供组件的可视化表示和在线编辑器，以供测试不同的内容和创建该内容集合的分享链接。

文档数据

文档数据是文档页用来创建初始视图的原始数据。

上述所有这些文件对于功能开发而言都是宝贵的产出，但其中有一个文件是整个流程的奠基石，因此也通常首先创建，那就是 JSON 模式。在这些产出中，没有哪个文件比 JSON 模式更好地描述了这项功能的需求和能力。每当一个组件的功能发生变化时，都是先由模式引起的，然后才扩展到组件的其他文件。正因为模式文件如此重要，所以这个开发流程称为*模式驱动的设计系统*。

10.2 模式驱动的设计系统

目前为止，在软件开发领域，*测试驱动的设计*是一种很常见且被广泛接受的开发方式。在动手编写具体的代码之前，我们先写好一套测试用例，用来描述该代码应该完成什么样的功能。刚开始开发时，难免会有测试不能通过，不过随着功能的不断完善，最终我们会使所有的测试都能通过。

*模式驱动的设计系统*的理念跟测试驱动的设计是相似的；区别在于，后者使用 NodeUnit 或 PHPUnit 编写测试用例来描述应用的正确功能，而前者则使用 JSON 来创建模式。这些 JSON 文件为设计系统的组件定义了正确的数据结构。在模式驱动的设计系统中，我们首先关注的是设计的内容和用户接口，而不是那些标记和 CSS。接下来看看 Red Hat 网站用到的一个简化版的 JSON 模式，了解一下这些信息如何帮助我们为设计系统做出更好的决定。

下面这个 JSON 模式定义了包含一个标题和两到三个 logo 图片的视图。

```
{
 "type": "object",
 "properties": {
   "headline": {
     "type": "string",
     "format": "text"
   },
   "body": {
     "type": "object",
     "oneOf": [
       {
         "title": "Two Up",
         "required": ["logos", "layout"],
         "properties": {
           "layout": {
             "type": "string",
             "enum": ["2up"],
             "options": {
```

```
                "hidden": true
              }
            },
            "logos": {
              "type": "array",
              "format": "tabs",
              "minItems": 2,
              "maxItems": 2,
              "items": {
                "$ref": "#/definitions/logo"
              }
            }
          }
        },
        {
          "title": "Three Up",
          "properties": {
            "layout": {
              "type": "string",
              "enum": ["3up"],
              "options": {
                "hidden": true
              }
            },
            "logos": {
              "format": "tabs",
              "type": "array",
              "minItems": 3,
              "maxItems": 3,
              "items": {
                "$ref": "#/definitions/logo"
              }
            }
          },
          "required": ["logos","layout"]
        }
      ]
    }
  },
  "required": ["body"],
  "definitions": {
    "logo": {
      "title": "Logo",
      "type": "object",
      "oneOf": [
        {
          "title": "Upload File",
          "properties": {
            "file": {
              "title": "Logo File",
              "description": "Upload your logo",
              "type": "string",
              "format": "file"
            }
          },
```

```
            "required": ["file"]
          },
          {
            "title": "Paste URL",
            "properties": {
              "url": {
                "title": "Logo URL",
                "description": "Paste a URL to your logo",
                "type": "string",
                "format": "url"
              }
            },
            "required": ["url"]
          }
        ]
      }
    }
  }
```

对于一个简单的组件来说，91 行的代码可能过于冗长了。尽管如此，我们还是先来仔细看看每个区块，然后再探讨一下它们各自如何帮助实现数据结构、编辑器 UI 和输出的 HTML。

第 2 行：

```
"type": "object",
"required": ["body"],
"properties" {
   "headline": {
   ...
```

我们创建的每一个模式都描述了需要收集哪些数据，这些数据最终会传递给模板或视图。模式中的每个属性要么代表单个数据，要么代表子模式里面的一组数据。因此在声明属性之后，这个模式文件会定义出一个键值对的对象，我们规定只有 `body` 字段是必选的。

第 5 行：

```
"headline": {
 "type": "string",
 "format": "text"
},
```

第一个属性 `headline` 是可选的，因为上面的必选字段数组中没有包括这一项。我们知道它是一个字符串，这说明了它在数据库中的存储方式。

我们还知道它的格式只是 `text`，这定义了用来收集标题的表单元素。此外，格式还可以是 `textarea`、`url`、`color`、`range`，或者任何其他有效的 HTML5 输入类型。虽然最后这些值都是存储成字符串，但是不同的格式可以提示编辑器以怎样的方式显示输入的数据。

第 9 行：

```
"body": {
 "type": "object",
```

第二个属性是 body，它的类型是 object，说明它不是一个单一的值，而是一组拥有各自的子模式的值。然而，这个例子中有两个可供选择的模式，下面就来解释一下。

logo 的设计要求我们提供能够适配两个或三个 logo 的布局。图 10-1 显示了一个样例。



图 10-1："three up"样例

如果有人以前见过类似的设计，他马上就会意识到这里缺少一个重要的信息。当这里只有两个 logo 时，布局会变成怎样？是保留原布局，让原本第三个 logo 的位置留空？还是换一个布局，让每个 logo 占据更大的空间？这两种选择将在图 10-2 和图 10-3 中分别表示。



图 10-2：相同的布局，少一个元素

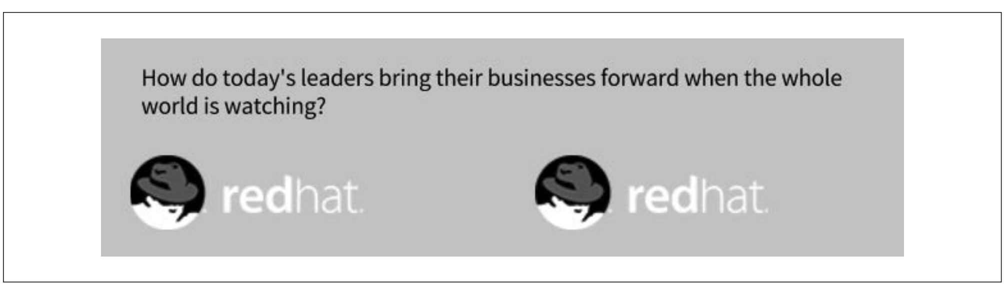


图 10-3：填满所有可用空间

我们的模式很快就能解决这个问题，这是一个强大之处。我们可以分别指定两个 logo 或三

个 logo 时的布局，实现的方法是定义两个不同的模式与之对应。我们将在第 11 行开始，为 `body` 的模式定义选项。

第 11 行：

```
"oneOf": [
  {
    ...
  },
  {
    ...
  }
]
```

JSON 模式提供了一些关键词，使我们的模式变得更灵活，包括 `oneOf`、`anyOf` 和 `allOf`。这里会使用 `oneOf`，意味着我们的数据需要符合下面模式的“其中一个”，而不是“全部”或“任何一个”。

正如你所见，`oneOf` 对应的是一个对象的数组。`body` 的数据需要与数组里面的一个（并且只有这一个）模式相匹配。

第 13 行：

```
"title": "Two Up",
"required": ["logos", "layout"],
"properties": {
  "layout": {
    "type": "string",
    "enum": ["2up"],
    "options": {
      "hidden": true
    }
  },
  "logos": {...}
```

第一个模式是“two up”。它有一个 `Two Up` 标题，当我们在 UI 层切换模式时可以使用这个标题。`required` 字段告诉这个子模式哪些属性是符合模式要求的必选项。这里声明 `logos` 和 `layout` 两个都是必选项。

第一个属性 `layout` 的独特之处在于，我们不希望用户设置它的值，但仍需要保存此值并传递给模板。因为它的值是必选项，因此符合模式要求的唯一方法就是传入 `2up` 这个字符串作为 `layout` 的值。

如果需求中要求允许用户选择想要的布局，那么可以用 `["2up", "3up"]` 来替换 `enum` 的值。因为已经决定不让用户选择布局，所以给属性的 `options` 项添加了 `"hidden": true`。这是一个在 UI 层隐藏此值，但仍然把它传递给模板的技巧。

既然模式的布局已经配置好了，现在转向希望用户上传的 logo。

第 23 行：

```
"logos": {
  "type": "array",
  "minItems": 2,
  "maxItems": 2,
  "format": "tabs",
  "items": {
    "$ref": "#/definitions/logo"
  }
}
```

logos 是另一个独特的属性，它的数据存储方式既不是字符串也不是对象，而是数组。既然是数组，那么就有几点需要确认。例如，需要确认数组中的最小值和最大值，如何在 UI 层格式化这个数组，以及数组中每一个元素所对应的模式。

对于 logos，我们设置 minItems 和 maxItems 的值都是 2。这说明我们唯一的选择就是传入两个元素，对我们的布局来说是一件好事，因为它就是接受两个元素的。

与提示 headline 属性应该作为 text 的输入类型来展示一样，这里指定 format 字段是标签布局。不过，这里的“标签”布局对 JSON 模式来说没有任何意义，这仅仅是一些提示，告诉我们可以将它传递给使用该模式的 UI。标签（tabs）代表什么是由每一个具体的实现决定的，因此如果 format 字段被忽略了，模式和产生的数据仍然是有效的。

最后定义 items 字段，这是一个让我们可以定义数组元素使用何种模式的关键词。在这个例子中，$ref 意味着正在使用一个 JSON 指针。这允许我们指向一个外部的定义，而不是在内部定义。#/definitions/logo 这个值指示指针去寻找当前模式中的根节点 #，然后返回 definitions/logo 中的对象，可以在第 60 行找到它。

这里使用一个引用值的原因是，“two up”模式和“three up”模式都是对一个 logo 使用相同的定义。这意味着我们可以创建两个模式，而只需指定那些变化的值。

此外，这里不限于内部引用，因为 $ref 还可以引用其他模式文件。这样就可以在单独的文件中创建常用的模式，然后在其他模式中访问它们。在这个例子中，我们选择使用一个本地引用，下面就来看看它究竟引用的是什么。

第 60 行：

```
"logo": {
  "title": "Logo",
  "type": "object",
  "oneOf": [...]
```

此处开始定义 logo 对象。我们给对象命名一个更可读的标题，即使只是一个首字母大写的

单词也好，然后规定对象的其余部分要符合下列模式中的一个（即 oneOf）。可以看到，我们想提供两种不同的输入 logo 的方法，而它们都需要拥有各自的模式。

第一种方法是直接上传文件，第二种是输入 URL。

第 64 行：

```
{
  "title": "Upload File",
    "properties": {
      "file": {
       "title": "Logo File",
       "description": "Upload your logo",
       "type": "string",
       "format": "file"
    }
  },
  "required": ["file"]
},
{
  "title": "Paste URL",
  "properties": {
     "url": {
      "title": "Logo URL",
      "description": "Paste a URL to your logo",
      "type": "string",
      "format": "url"
    }
  },
  "required": ["url"]
}
```

此时就可以设置 oneOf 这个区块里的标题，包括 file 这个属性的标题。这里有一个重要的值，就是 file 属性的 format 值，它将通知 UI 提供一个“选择文件”的对话框。但是如果用户希望粘贴一个 URL 进来，那么可以切换到第二个模式，它提供一个 URL 字段，并包括必要的帮助文本。

传递到Twig文件

流程的最后一步是把所有的信息传递到模板文件。首先看看有效的数据是什么样的：

```
{
  "headline": "This is my headline",
  "body": {
    "layout": "2up",
    "logos": [
      {
        "url": "http://www.fea.pub/my-logo.png"
      },
      {
```

```
          "file": "path/to/my-other-logo.png"
        }
      ]
    }
  }
```

我们的 Twig 模板大概是这样：

```
<div class="logo-wall">
  {% if headline %}
    <h1 class="logo-wall-headline">{{headline}}</h1>
  {% endif %}
  <div class="logo-wall-logos" data-layout="{{body.layout}}">
    {% for logo in body.logos %}
      <img class="logo-wall-logo"
       src="{{ logo.file ?: logo.url }}" />
    {% endfor %}
  </div>
</div>
```

我们从模式中可以获取到的第一个信息就是 headline 这个属性。我们不仅知道它的字段名叫 headline，还知道它的值是可选的。因此，我们在 Twig 模板中先判断一下 headline 的值是否存在，如果存在，那么在 H1 标签里把它打印出来。

再看一下模式的主体部分，我们使用布局的值来设置 data-layout 这个属性。在 CSS 中实际执行的逻辑随后会处理，不过目前可以肯定的是，如果数据源中含有三个图片，那么 body.layout 将会是 3up，而如果只有两个，那么它会是 2up。

最后，我们知道 logos 属性是一个对象的数组，包括 URL 字符串或文件路径字符串。Twig 模板提供了一个简单的方法来判断 logo.file 是否存在：如果存在，那么使用它；如果不存在，那么使用 logo.url。

因此，这个模式最后提供了以下内容：

- 定义所有可能字段中的内容范例及其数据类型和建议的输入类型，以及是否为必选项
- 在把数据传递给模板之前，验证其有效性
- 对于创建模板时数据中可能出现的所有变量，给出了一系列所需的定义
- 定义了用户输入数据时的用户界面，图 10-4 展示了使用 JSON 编辑器解析模式（https://github.com/jdorn/json-editor）的一个例子

这就是 91 行 JSON 代码带来的众多益处。对于正在创建的设计系统而言，这是一个传递理念和想法的完美工具，否则要用大段的文字才能阐述清楚。

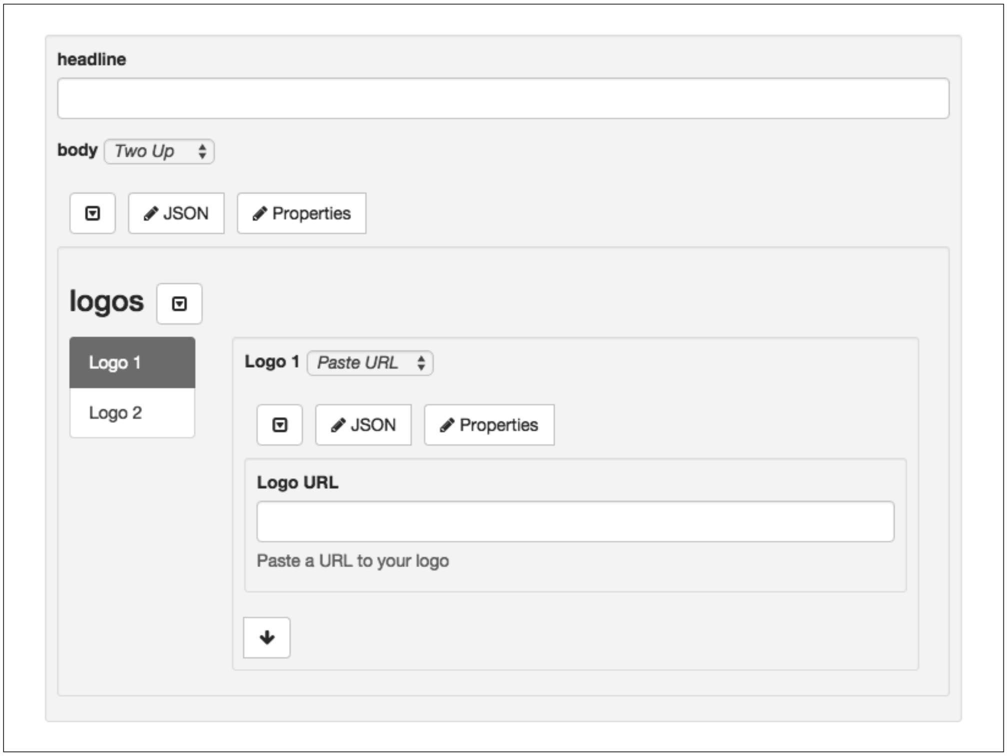


图 10-4：Jeremy Dorn 的 JSON 编辑器

用这个系统工作的每一天，我都惊讶于这些模式能够如此方便地对模板进行改动或增加内容，提供给开发人员的交互方式也非常好用。这些模式免去了很多过去逐级交付流程中的主观猜测。

通过遵循公共的标准，我们一直在寻找可以给设计系统带来价值和功能的第三方工具。对模式系统的坚持所带来的回报是多方面的，以至于我确信自己以后再也不会创造不带模式的设计系统了。这就是模式驱动的设计系统的魅力。

第四部分

测试核心

前端架构师的重要职责之一就是整体把控网站和设计系统的开发。然而，任何参与过大型项目的开发人员都很清楚，自己单枪匹马去兼顾项目的方方面面是不可能的。在人数多达几十个、每周提交上百次、代码多达上千行的团队中，仅靠一个人根本无法评估每一段写入系统的代码所产生的影响。然而，这种级别的评估正是我们需要的。

新代码给系统带来的问题涉及方方面面。有些可能会影响系统的计算结果，导致错误的产品价格；有些则可能影响用户使用购物车结算的功能；还有一些问题会影响网站的视觉表现，虽然功能仍然可用，但是不完整或不一致的界面影响了用户体验；最后，就算用户界面是完整的，购物车功能也正常，但在某些特定设备上或特定地区访问网站时，结算流程可能还是无法顺利完成。虽然这些不同类型的问题源于代码库里不同的部分，但它们导致的结果是一样的：销量下滑。幸运的是，它们都有同样的解决方法：测试！

虽然作为架构师不可能把所有时间都花在检查每一行提交到系统的代码上，但我们可以使用各种测试工具来验证应用程序是否正常工作。

当你开始为应用程序规划测试时，请记住以下几条建议。

- 测试用例应该在建站的同时，甚至在建站之前就开始编写。
- 测试代码是真实的代码，应该一起或立即提交到系统代码库中。
- 必须在所有的测试用例都通过之后，才能把代码合并到主干中。
- 在主干上运行测试工具，结果应该都为通过。

测试意味着那些对系统计算结果、进行重要业务操作、渲染正确界面或者提供流畅体验产生不良影响的代码不会再被合并进去。因此，架构师与其尝试检查成千上万行代码，不如去关注构建高质量的系统和完整的测试。

第11章

单元测试

单元测试是将应用程序分解为尽可能小的函数，并创建可重复的、自动化的测试用例的过程。在同等条件下，这些测试用例应该一直产生相同的结果，它们是应用程序的灵魂，并为今后所有应用程序的代码提供构建的基础。如果没有单元测试，不常使用的函数可能长达数月都不会被发现有 bug；相反，通过使用单元测试，我们可以在任何代码合并到主干之前就验证每个系统函数的功能，不会等到代码实际应用到产品中时还会出现问题。

前端架构师的主要责任是保证开发者拥有尽可能高效的开发工具。单元测试就是其中之一，它可以用于构建任何规模的应用，无论构建应用程序逻辑的主要语言属于前端语言还是后端语言，你都有大量的工具可以应用到工作流中。无论使用的是 PHP 的 PHPUnit、Node 的 NodeUnit，还是 JavaScript 的 QUnit，你都能找到用于构建单元测试的成熟稳定的平台。

虽然技术栈（和与之相关的测试）有可能留给软件架构师来决定，但前端开发人员写出的代码可能也需要测试。因此，熟悉尽可能多的测试工具很重要。掌握所有工具，甚至达到精通的程度，那是非常奢侈的。但如果对基本概念有着深刻的理解，将能帮助你和你的团队编写出更加可测试的代码，并快速地掌握任何测试框架。

下面首先来回顾一下这些基本概念，然后再来看一些实际的代码。

11.1　单元

“一次只做一件事，并把它做好”是构建基于单元测试的应用程序的原则。我们在写函数时经常想同时实现很多功能，结果最后不仅降低了效率，还增加了测试的难度，因为这样的函数无法复用。

思考一个简单的函数：通过客户的地址，计算出将产品从最近的分拨中心运输给客户的运费。

下面来分解一下这个函数。函数要做的第一件事是通过客户的地址找到最近的分拨中心；然后使用分拨中心的地址，计算出分拨中心到客户地址的距离；最后，使用这个距离，计算出将货物从点 A 运输到点 B 的费用。因此，虽然只有一个函数，它却做了 3 件相互独立的事：

(1) 根据地址找到最近的分拨中心
(2) 计算两个地址之间的距离
(3) 根据距离计算运费

回顾一下“一次只做一件事，并把它做好”的理念，很明显可以通过 3 个独立的函数更好地实现通过地址来查询运费的功能。当我们对功能进行拆分后，就可以得到 3 个函数：一个根据地址找到最近的分拨中心，一个计算两个地址之间的距离，还有一个根据距离计算运费。

11.1.1　更多重用

现在，这 3 个函数可以在整个应用中使用了，而不只是用来计算运费。如果程序的其他部分需要找到最近的分拨中心或者计算两个地址之间的距离，可以直接调用这些已经写好的函数。现在，在较大的函数体中，我们不用一遍又一遍地重写功能相同的模块了。

11.1.2　更好的测试

我们可以测试每个独立且可重用的函数，而非测试应用程序所能计算的每一条运输路线。随着应用程序的发展，开发一个新功能所需要创建的新函数越来越少。最终，我们用较少的低复杂度的函数完成了高复杂度的功能。

11.2　测试驱动的开发

大多数人在首次接触单元测试时，可能写过一些功能代码以满足业务需求（比如上文中的计算运费的例子），然后也努力地将它重构为更小的、可重用的、可测试的代码，之后才去思考如何写测试用例。测试驱动的开发（test-driven development，TDD）则颠倒了这一思路，它将单元测试放在第一位，之后才是编写功能代码。

但如果为还没有创建的函数写测试用例，那些函数岂不是肯定无法通过测试吗？的确！但测试驱动的开发的目标是，通过测试用例来描述一个正确编写的系统应如何工作，并为实现这个系统来铺平道路。

对于上文提到的计算运费的例子，在测试驱动的开发中会为三个实现业务需求的函数分别编写单元测试。开发者的工作就是将一开始没有通过单元测试的函数变为能够通过测试的函数。首先实现一个可以正确地查找出最近的分拨中心的位置的函数，并通过第一个单元

测试。然后，继续实现计算距离和计算运费的函数，最终得到三个通过与之对应的单元测试的函数。

完成上述步骤后，我们不仅实现了应用的计算运费的功能，还让这些函数有了完全的测试覆盖率。

11.3 一个测试驱动的例子

单元测试的核心理念其实非常简单。它的基本思路是调用要测试的函数，传递一些预先设置好的参数，并描述结果应该是什么。下面来看看如何为通过给定距离来计算运费的函数设计功能测试：

```
function calculateShipping(distance) {
  switch(distance) {
    case (distance < 25):
      shipping = 4;
      break;
    case (distance < 100):
      shipping = 5;
      break;
    case (distance < 1000):
      shipping = 6;
      break;
    case (distance >= 1000):
      shipping = 7;
      break;
    }
    return shipping;
  }

QUnit.test('Calculate Shipping',function(assert) {
  assert.equal(calculateShipping(24),4,"24 Miles");
  assert.equal(calculateShipping(99),5,"99 Miles");
  assert.equal(calculateShipping(999),6,"999 Miles");
  assert.equal(calculateShipping(1000),7,"1000 Miles");
};
```

QUnit（https://qunitjs.com/）有多个用于测试断言的操作符，包括用于测试布尔值的 `ok()` 和用于比较复杂对象的 `deepEqual()`。在这个例子中，我们使用 `equal()` 来比较 `calculateShipping()` 的返回值与期望值。只要 `calculateShipping(24)` 返回的是 4（这个例子中确实将返回 4），这个单元测试就能通过。第三个参数——`24 Miles`——用来在测试输出中标记相应单元测试的结果。

上文中通过距离计算运费的单元测试（当然也包括其他函数的单元测试）就绪后，我们就掌握了一个测试集，通过运行该测试集可以确认系统能否正常工作。如果有人要改 `calculateShipping()` 的函数名或者运费，这个测试集就会反馈提示失败，因此我们能够在有问题的代码部署到生产环境之前就修复它们。

QUnit 的功能远远不止上文例子中提到的。举例来说，它可以测试同步函数，也可以测试异步函数；它还可以与单元测试中加载的 Web 页面进行交互（其实用 JavaScript 都能实现）。因此，如果你的单元测试包括用户点击鼠标或按下键盘时的返回值，QUnit 也能够支持。

11.4　测试覆盖率要多大才足够

确定合适的测试覆盖率是很难权衡的一件事情。如果你不是在进行测试驱动的开发（这种开发中，没有代码是不需要单元测试的），那么确定测试覆盖率将非常重要。如果测试所有的代码，开发进度可能停滞不前；而如果测试不够，就有漏掉新问题的风险。

11.4.1　解决分歧点

如果为已有的项目设计单元测试，大部分情况下，你可能没有时间或者预算为现有的功能编写 100% 覆盖率的测试集。但这不是问题！测试覆盖率的好处是，即使一个单一的测试也能够为系统建设贡献价值。因此，在决定从哪里开始写单元测试时，可以从能够获得最大收益的地方开始。有时候，最大的收益就是为系统最简单的部分编写单元测试。就像在处理更重要的债务之前，先还清小额信用贷款一样，写一些简单但仍有价值的测试用例是为单元测试造势的好方法。

一旦有了能提供基本覆盖率的测试集，就可以开始寻找系统中最关键的部分，或者过去频繁出问题的部分，在需求列表中为它们分别创建需求，并确保尽快推动这些需求。

11.4.2　从测试覆盖率开始

如果有幸作为前端架构师启动全新的项目，你的工作就不仅仅是设置好测试框架了，还要确保开发流程本身为单元测试做好了准备。就像写文档和进行代码审核一样，写单元测试也要花不少时间。你需要确保任何需要测试的需求都有额外的时间来编写单元测试，并且确认所需的测试覆盖率。

在 Red Hat 公司，每个用户故事都始于一系列的任务，包括用于开发并验证该功能所需的测试覆盖率，并为此预留了时间。如果一个新功能需要花费 8 个小时开发完成，我们要确保另外预留 2 个小时来编写用例并验证测试覆盖率。预留的时间通常很难争取，因此前端架构师通常需要扮演起外交人员和销售人员的角色。尽管这样会多花费 25% 的时间，但我们知道这其实会节省很多后续回头追查 bug 的时间。

正如前面所说的，并不是所有的功能都需要同样的测试覆盖率。但前提是，每一个用户故事都是以测试覆盖率的相关任务作为开始的。只有当所有人都认为给这些任务写测试用例没有必要时，才考虑去掉它。这样我们才能确信，对于任何需要测试的功能，都已经安排了足够的时间去完成它们。

第 12 章

性能测试

任何测试的目的都是为了避免不流畅的用户体验，而糟糕的网站性能正是导致用户体验不流畅的主要原因之一。因此，性能测试虽然不是针对系统或视觉问题的测试，却也是测试库的重要组成部分。

性能测试衡量的是影响用户使用网站的流畅程度的关键指标，包括页面大小、请求数量、首字节时间（time to first byte，TTFB）、加载时间和滚动性能。

性能测试的关键是制定合适的预算并坚持下去。如何进行这一步，将决定你的项目测试的有效性。

12.1　制定性能预算

制定性能预算是指为每个关键指标设定目标值，然后在所有代码合并或部署之前持续测试这些指标。若有任何一个指标没通过测试，则需要调整新增的功能，或删除一些其他功能。

正如财务预算那样，很少有人对性能预算的前景感到非常兴奋。对大多数人来说，预算意味着花费更少、获得更少、乐趣更少……总之就是“更少”。在一个一直告诉我们可以拥有更多的世界里，“更少”实在是没什么意思。作为设计师，如果不能肆意地折腾高分辨率图像和全屏视频，就会觉得自己的创造力被扼杀了。作为开发者，如果没有 CSS 框架、几个 JavaScript 框架和一些 jQuery 插件，就会觉得自己无法工作。“更少”一点都不好玩！

作为一个在过去四年里一直按照财务预算生活的人，我很理解没有得到自己想要的一切意

味着什么。但从好的方面来看，当我真的按照预算花了一大笔钱时，我不会因此有愧疚感或负债。同理，做好预算让我们能够负责任地“消费”并且不会后悔。

正如财政纪律和财务预算那样，用户体验准则和性能预算可以帮助我们实现最终目标，即创建一个性能良好并且对用户有吸引力的网站。

财务预算通常基于一个人的收入，而性能预算更多地与外部因素相关，而不是内部因素。

12.1.1 竞争基线

制定性能预算的一种方法是参考竞争对手。虽然“至少我比某某更好”不能作为网站性能不佳的借口，但是这种方法确实可以保证你有一定的竞争优势。

首先来看几个竞争对手的主页和登陆页，然后和自己的网站比较加载时间、网页大小和其他的关键指标。你的目标不是达到竞争对手的水平，而是要保证领先竞争对手 20% 甚至更多。因此，如果竞争对手的产品列表页在 3 秒内加载完成，那么你就要确保自己网站的产品列表页在 2.4 秒内加载完成，甚至更快。这些超越竞争对手 20% 的优势，是用户将你和竞争对手区分开来所需要的。

优化关键指标并不能一劳永逸，它需要的是持续监控。可以确定的是，你的竞争对手也正在寻找方法来改善和优化他们自己的网站。如果他们也一直参考你的网站来确定预算，那么你其实也在推动他们减少性能预算！

12.1.2 平均基准

不管你的竞争对手是谁，把你的网站性能基线与行业平均水准和通用的最佳实例相比较总是必不可少的。我们没有理由因为竞争对手的落后而保持平庸。

HTTPArchive（http://httparchive.org/）是个不错的服务，它测试并记录了几十万个网站的各种性能指标，截至 2015 年 4 月，有几个值得注意的数据。

- 页面大小：2061KB。
- 总请求次数：99。
- 可缓存资源所占比例：46%。

因此，如果想让自己的网站比大部分网站都快，可以考虑设定一个目标：一个 1648KB 的网站，包括 79 个请求，其中 44 个请求可以缓存。这将使你的网站领先平均水平 20%。

现在我们已经知道几个制定预算的方法了，那么当我们开始测试时，需要考虑的预算项目有哪些呢？

12.2 原始指标

网站性能最基本的测试是看渲染页面所需要的资源，包括这些资源的大小和总数。

12.2.1 页面大小

网站页面正变得越来越大。从 2014 年 4 月到 2015 年 4 月，网站页面的平均大小从 1762KB 增长到 2061KB，以每年 17% 的速率增长。而回到 2011 年 4 月，网站页面的平均大小只有不到 769KB！

虽然页面大小并非影响网站加载速度的唯一因素，但它确实对此影响重大。页面过大还有其他副作用，我们知道越来越多的用户通过移动设备访问网站，而他们要为数据流量付费。你的网站页面越大，用户就将支付越多的费用，尤其是在发展中国家。不然试试访问 What Dose My Site Cost（http://whatdoesmysitecost.com/）来看看新的图片轮播功能让德国的手机用户花了多少钱。

当你希望缩减页面的大小时，可以从以下几个显而易见的地方开始。

- 图片占据页面平均大小的 61%。
  - 优化 PNG 图片，并降低 JPEG 图片的质量。
  - 利用新的响应式的 `<picture>` 标记和 `srcset` 属性来下载大小合适的图片。
  - 制定一个预算，如果没有移除任何图片，就不增加图片大小。

- 太多自定义字体很快会使网页变得臃肿。
  - 制定一个字体预算，不考虑增加第二种或第三种字体。
  - 考虑必要的字体粗细，因为每增加一种粗细变化，都会使字体文件增加几千个字节。
  - 虽然图标字体很不错，但要注意文件大小，因为图标字体会使字体文件迅速变大。如果一个网站只使用字体文件的一部分，其他网站使用另外的部分，那就拆分字体文件。也可以考虑使用内联 SVG 代替图标字体，只加载需要的 SVG 就可以得到很多图标字体了。

- JavaScript 框架、jQuery 插件和 CSS 框架常常使页面大小增加很多，却收效甚微。
  - 很多网站都已远离 jQuery，因为 vanilla JS 就可以满足其需求，尤其是网站只针对现代浏览器时。
  - jQuery 插件虽然可能会提供一些很厉害的功能，却常常使页面大小显著增加。考虑在现代浏览器上使用 CSS 能否达到同样的效果，并在低版本浏览器上合理地回退。
  - 像 Angular 或者 Ember 这样的大型 JavaScript 框架也许能完成你的工作，但生成的网页大小会超出实际工作的需要。如果只需要使用 Angular 的视图层，那么最好使用 React 或 Mustache 来替换。

- ♦ CSS 框架往往是乱七八糟的。它包含可能会用到的所有想象得到的样式。虽然这对于网页快速成型很有帮助，但从现有的几千字节的 CSS 和 JavaScript 出发来构建网站，会让你在开始写第一行代码之前就陷入困境。

- 使用压缩。
  - ♦ JavaScript 可以在构建流程中以编程的方式进行压缩，而且可以在服务器将文件发送到浏览器之前使用 gzip 压缩。这些都是缩减网页大小的关键步骤。

12.2.2 HTTP请求次数

浏览器对页面渲染所需的每个文件都要进行 HTTP 请求。因为每个浏览器对 HTTP 请求的次数有单域名并发限制，所以大量单独的文件意味着浏览器必须进行多轮并发请求。在速度较慢的网络环境中，这么多并发请求会造成很复杂的影响。因此，减少获取所需文件的并发请求次数，效果会很显著。

可以通过如下的方法减少并发请求次数。

- 减少 HTTP 请求的次数。
  - ♦ 不要提供数十个单独的 CSS 文件和 JavaScript 文件，而是把它们合并到一个文件中。
  - ♦ 把多个单独的图像文件合并成一个图像映射或者图标字体。有很多不错的工具（例如 Compass 和 Grunt/Gulp 插件）可以帮你自动化地完成这些任务。
  - ♦ 延迟加载页面最初加载所不需要的资源。这可能是直到用户与页面交互才需要的 JavaScript 文件，也可能是初始加载窗口之下距离较远的图片。

- 增加浏览器每次并发请求的资源个数。
  - ♦ 分拆你的资源到不同的服务器（或者 CDN），可以使得浏览器单次并发下载更多的资源，因为浏览器的并发请求数量限制是针对单个服务器的。

12.3 计时度量

除了站点的资源数量和大小，还有其他的计时度量会影响用户对网站性能的体验。

首字节时间

　首字节时间是指从浏览器请求网站页面开始，到浏览器接收到第一个字节之间的毫秒数。这个数值用来测量浏览器和服务器之间的连通路径，包括 DNS 查询、初始连接和数据接收。它并不是判断站点性能的最佳标准，却是一个值得关注的指标。

开始渲染时间

　更有价值的计时度量是“开始渲染时间”。这个度量是指用户开始在页面上看到内容的时间。这意味着所有阻塞渲染的文件都已经加载完成，浏览器已经开始渲染文档模型了。可以通过以下方式优化开始渲染时间：延迟加载阻塞渲染的 JavaScript 和 CSS 文

件、将关键的 CSS 代码内联到页面头部、用数据 URI 代替图片资源，以及延迟加载所有在文档模型渲染完成后才下载的资源。

文档完成时间

只要最初请求的资源已经加载成功，就可以认为文档“完成”了。文档完成时间不包括 JavaScript 中拉取资源消耗的时间，因此延迟加载的资源不会影响这个指标。

12.4 混合度量标准

混合度量标准不是度量离散的值，而是根据多个性能指标综合打分得出。

12.4.1 PageSpeed分数

PageSpeed（https://developers.google.com/speed/pagespeed/insights/）是 Google 开发的网站工具和 Chrome 浏览器的扩展程序，用来分析站点的性能和网站的可用性，它给出一个用百分比表示的分数，并解释了提高分数的方法。测试包括：

- 是否存在阻塞渲染的 JavaScript 或者 CSS
- 重定向至登陆页
- 图片优化
- 文件压缩
- 服务器响应时间
- 服务器端压缩
- 浏览器端缓存
- 点击目标的大小
- 窗口可见区域的配置
- 清晰的字体大小

12.4.2 Speed Index指标

根据 Speed Index 项目主页上的描述，Speed Index（https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index）指的是页面可见部分展示完成的平均时间，该指标通过用毫秒表示，并取决于视图端口的大小。

混合度量标准的分数考虑了上述多个单一的度量标准，并将这些标准和页面加载时用户可以实际看到的标准结合起来。Speed Index 是度量终端用户实际体验的最好标准之一。

12.5 设置性能测试

现在我们已经知道可以测试哪些指标，以及如何制定性能预算。那么接下来看几个用于自

动化测试流程的方法。不管是测试站点是一个还是几十个，恐怕没有人愿意手动测试。

12.5.1 Grunt PageSpeed插件

我们要了解的第一个测试流程自动化工具是 Grunt 的 PageSpeed 插件（https://www.npmjs.com/package/grunt-pagespeed）。顾名思义，这个 Grunt 插件可以自动化地运行 Google 的 PageSpeed 来测试我们的站点。我们能在所有的分支合并请求和后续集成构建之前就运行该 Grunt 任务 , 而不用将你的 URL 插入到 Page Speed 页面中或使用 Chrome 扩展。

为了设置 Grunt PageSpeed 插件，首先使用标准的命令安装和引入该插件。

```
$ npm install grunt-pagespeed --save-dev

// 加入Gruntfile.js
grunt.loadNpmTasks('grunt-pagespeed');

// 在Gruntfile.js中加入grunt.initConfig
 pagespeed: {
  options: {
    nokey: true,
    url: "http://redhat.com"
    },
    desktop: {
      options: {
        paths: ["/en", "/en/services"],
        locale: "en_US",
        strategy: "desktop",
        threshold: 80
      }
    },
    mobile: {
      options: {
        paths: ["/en", "/en/services"],
        locale: "en_US",
        strategy: "mobile",
        threshold: 80
      }
    }
  }
```

使用上述代码，我们可以使用 base URL 中的一系列页面（这个例子中是 http://www.redhat.com/）自动化地运行桌面端和移动端的测试。只有得分大于 80，测试才会通过；如果得分小于 80，测试就会失败，意味着我们需要继续修改，以达到阈值。

12.5.2 Grunt Perfbuget插件

另外一个很好的 Grunt 工具是 Grunt Perfbudget（https://github.com/tkadlec/grunt-perfbudget）。该 Grunt 插件集成了 Marcel Duran 的 WebPageTest API（https://github.com/marcelduran/

webpagetest-api），让我们能以编程的方式拉取 WebPageTest（http://www.webpagetest.org/）的结果，并将其与我们设置的预算相比较。如果你还没使用过 WebPageTest，你一定会喜欢上它的。WebPageTest 可以通过模拟不同的网络连接类型和全球的地理位置来测试网站的多个指标。我不会详细介绍 WebPageTest 的每一项功能，但是花五分钟看一下你的网站的测试结果，我确信你会喜欢上 WebPageTest 提供的一系列丰富的信息。

那么一起来看看如何在 Grunt 中配置它吧：

> 目前可以在 WebPageTest 的网站（http://www.webpagetest.org/getkey.php）上申请 API 密钥，但它的使用范围有限。

```
$ npm install grunt-perfbudget --save-dev
// 加入Gruntfile.js
grunt.loadNpmTasks('grunt-perfbudget');

perfbudget: {
  default: {
    options: {
      url: 'http://www.redhat.com/en',
      key: 'SEE_NOTE_ABOVE',
      budget: {
        visualComplete: '4000',
        SpeedIndex: '1500'
      }
    }
  }
}
```

上述设置使我们可以在 Red Hat 主页上自动运行全部的 WebPageTest 测试集。在这个例子中，我设置视觉完成时间指标为 4000 毫秒，而 Speed Index 指标为 1500 毫秒。如果有任何测试返回结果超出预算，那么就会得到一个严重错误提示信息，提示我们重新审视最新提交的代码，检查究竟是什么导致超出预算。

12.6 小结

通过适当的自动化测试流程和有竞争力的预算，你就有了一个很好的起点，在此基础上可以持续开发新的功能并改进网站，同时确保提交的改变不会超出预算。

第 13 章

视觉还原测试

告诉我，这个场景是不是很熟悉：最近几周以来，你一直在做网站的联系表，努力地微调表单的各种输入框，直到它们看起来和 Photoshop 设计稿完全一样。你已经小心翼翼地对比了所有的内边距、外边距、边框和行高。而且产品负责人也认为，“以后用这个联系表就可以了”。安全地提交了代码之后，你开始下一项工作，并且不再做那些重复出现的、有关浏览器下拉列表渲染差异的噩梦。

几个星期后，你在任务列表中惊讶地发现一个熟悉的条目：联系表。一些设计师、业务分析师或质量保证工程师带着尺子来到你的设计前，把它和最新的 Photoshop 设计稿相比较，发现了一系列差异。

但是，为什么呢？难道有人破坏了你的代码？或者有人修改了设计稿？追查罪魁祸首对你来说太奢侈，你没有那个时间。因此，你坐下来开始了一系列的调整，并且希望这是你最后一次碰这个联系表，但同时也已经向现实屈服：在网站发布之前，你可能还要再碰到它几次。

13.1 常见的质疑

在这个世界上，我最喜欢的声音就是决策者的尖叫：“功能 X 被完全破坏了！”翻译为开发者的术语，这通常意味着一些字体样式不正确，或者一些垂直对齐需要修复。这个功能本身有没有被确认或达成一致并不重要，重要的是当前产品的样子和过去一周决策者一直关注着的 Photoshop 设计稿有区别。

由于这些在我身上也发生了一遍又一遍，下面就来探索一下导致这个问题的常见原因，以及究竟是什么让你像旋转木马一样陷入死循环。

13.1.1 不了解情况的开发者

即使你的代码完美无缺，也会被其他开发者的短短几行不正确的代码破坏掉。其他开发者一直在做别的表单组件，并没有意识到他们正在编写的 CSS 类是和你的联系表单共用的。他们可能在你提交代码后的几周内编写了这些代码，也可能是和你同时写出的。

无关的页面上的小样式的改动，往往会被 QA 团队忽略。他们手上有几十个甚至上百个页面需要测试，因此没办法注意到标签字体上只有两个像素的改动。

13.1.2 不一致的设计

请允许我告诉你一个 Photoshop 的阴险的小秘密。当设计师改变了某个文件中的表单标签字体时，其他 PSD 文件中的字体并不会奇迹般地同步更改。可悲的是，在 PSD 文件中没有哪一页是用来级联地说明所有元素的样式规范。即便所有的设计师都知道字体更改并修改了 PSD 文件，但那些遗留在邮件列表、BaseCamp 会话和 Dropbox 目录里的 PSD 文件依然是修改前的模样。

如果有设计师、业务分析师或质量工程师，对照着不知哪个版本的 PSD 文件审核了你的联系表，十有八九他们哪天会发现你的联系表是有问题的（也就是他们认为的“功能被完全破坏了”）。然后他们给你创建一个新的用户故事去解决这些问题，而你只能祈祷这些改变不会在下一次某个设计师看到联系表时给你添更多的麻烦。

13.1.3 举棋不定的决策者

根据无限可能性法则，如果有足够多的决策者仔细研究足够多的功能，那么百分之百会有人发现想要修改的地方。

改动是不可避免的，如果使用恰当的开发模式，那么改动也是完全可以接受的。然而，当改动伪装成缺陷时（或者二者之间也不存在什么区别），开发者最终将花费大量的时间来开发还处于原型期的功能上。

在公开发布前对功能进行原型开发是没有问题的，事实上这是很好的实践。但原型需要基于设计快速迭代之后的最终设计稿做出，最终结果是一个大家一致认可的产品。如果让开发者在每个冲刺（sprint）周期里开发原型，又不停地在下一个冲刺周期里修改它，这不仅会降低开发者的效率，也是一种极为低效的原型开发方法。

13.2　一个经过测试的解决方案

虽然上述每个场景都突出了更深刻的组织层面的问题，但它们都可以通过适当的测试覆盖率来缓解。我们不去测试 JavaScript 函数的有效返回结果，而是抓取已授权的设计系统的视觉外观，从而验证我们没有偏离该系统。在提交之前抓取这些视觉还原是保证设计系统一致性的关键。

视觉还原测试让我们可以将正在开发的版本或者即将部署的版本（新版本）与正确的版本（基线版本）进行视觉对比。这个过程只不过是抓取基线版本的截图，与最新版本进行对比，并找出像素层面的差异。

通过把这些基线图片提交到仓库，或者在测试库里将其标记为通过，我们就对任何特定的功能（在我们的例子中为联系表）在像素级别的视觉表现有了签名确认并一致认同的核对记录。在任何代码提交到主分支之前，视觉还原测试提供了一种测试网站所有功能的方法，以确保没有出乎意料的视觉改变。

这样，我们对一些原本由 PSD 文件不一致导致的 bug 报告也有了应对措施。借助于已经签名并提交到代码库的基准，我们可以跑一遍测试程序，并自信地回复道："我们的代码没有问题，一定是 Photoshop 文件出了问题。"同样，我们也可以借此分辨真正的 bug 和新的变更需求。

13.3　视觉还原测试的多面性

借助于多种技术和流程，视觉还原测试可以有多种风格。虽然新的工具不断地被发布到开源社区，但它们通常是一小部分功能的组合。大多数工具可以归属为以下几类。

基于页面的比较

　Wraith（https://github.com/BBC-News/wraith）是一个基于页面的比较的例子。它使用 YAML 作为设置文件，因此可以很轻松地比较来自两个不同来源的一大串页面列表。当你不期望两个不同来源的页面有任何差异时，比如需要比较线上页面和在工作中即将部署的页面时，这个方法会很合适。

基于组件的比较

　在基于组件或基于选择器的比较方面，BackstopJS（https://github.com/garris/BackstopJS）是一个绝佳的选择。基于组件的比较工具使你可以抓取独立的页面片段进行对比，这样可以写出更有针对性的测试，并防止误报。举例来说，如果页面顶部的组件把所有其他组件都挤了下去，那么所有其他的组件就都被误报了。

CSS单位测试

Quixote（https://github.com/jamesshore/quixote）是一类独特的比较工具，用于比较 CSS 单位的差异而不是视觉上的差异。Quixote 可以设置 TDD 模式的测试用例，这些用例会设置好预期的 CSS 数值（比如字体大小为 1em，侧边栏的内边距是 2.5%），然后检测页面是否满足这些条件。它还可以诊断页面是否遵守品牌的视觉规范，比如 logo 的尺寸是否正确，以及 logo 与其他内容是否保持恰当的距离。

基于无头浏览器的测试

Gemini（https://github.com/gemini-testing/gemini）是一款可以使用无头浏览器 PhantomJS（http://phantomjs.org/）的比较工具，它可以在抓取截图之前加载 Web 页面。PhantomJS 是 JavaScript 实现的 WebKit 内核的浏览器，这意味着它速度非常快，并且具有跨平台的一致性。

基于桌面浏览器的测试

Gemini 非常独特，它支持在传统的桌面浏览器上运行测试用例。为了达到这个目的，Gemini 使用 Selenium 服务器（http://docs.seleniumhq.org/download/）打开并操作系统中安装的浏览器。这种方式没有基于无头浏览器的方式快，而且也受到系统安装的浏览器版本的影响；但是它更接近真实情况，并且可以发现某个特定浏览器引入的 bug。

包含脚步库文件

CasperJS（http://casperjs.org/）是一个导航脚步库，可以和 PhantomJS 等无头浏览器协同工作。该工具可以和在浏览器中打开的页面进行交互。使用它，你可以点击按钮，等待模态窗口，填充并提交表单，最终对结果进行截图。CasperJS 还可以在 PhantomJS 打开的页面中执行 JavaScript，你可以隐藏元素、关掉动画，甚至可以用静态模拟内容替代动态真实内容，以避免由于"最新博文"的发布导致测试不通过。

基于图形用户界面的比较工具，支持更改确认

Diffux 项目（https://github.com/diffux/diffux）存储了测试历史数据，并可以在基于 Web 的用户界面中提供测试结果的反馈。基准图像存储在数据库中，任何对它的改动都必须在该应用界面中标记为接受或拒绝。当不懂技术的股票持有人需要确认每一个变动是否正确时，这些工具就显得很重要了。

基于命令行的比较工具，支持更改确认

PhantomCSS（https://github.com/Huddle/PhantomCSS）是一款基于组件的比较工具，借助于 PhantomJS 和 CasperJS，它可以仅通过命令行来运行。测试是通过命令行终端运行的，无论测试是否通过，其结果都会输出到命令行终端里。这种类型的工具尤其适合通过 Grunt 或者 Gulp 运行，而其输出也很适合 Jenkins 或者 Travis CI 等自动化工具。

下一章将介绍 PhantomCSS 以及如何将其集成到你的项目中，同时会涉及 Red Hat 所使用的具体测试方法。

第 14 章

Red Hat测试方法

14.1 实践视觉还原测试

过去几年，我一直使用 PhantomCSS，因为它提供基于组件的对比方法，并且它使用的无头浏览器和脚本库可以很方便地集成到我当前的系统构建工具中。那么，接下来看看 PhantomCSS 的设置方法，以及它在 Red Hat 公司中是如何使用的。

14.1.1 测试工具集

PhantomCSS（https://github.com/Huddle/PhantomCSS）是由下列三种工具组成的强大合体。

- PhantomJS（http://phantomjs.org/）是基于 WebKit 内核的无头浏览器，它让你可以快速地渲染页面，而更重要的是，还可以对页面进行截图。
- CasperJS（http://casperjs.org/）是一个导航和脚本工具，通过它你可以与 PhantomJS 渲染出的页面进行交互。我们能滑动鼠标、单击、在文本框中输入文字，甚至在 DOM 节点上直接调用 JavaScript 函数。
- ResembleJS（http://huddle.github.io/Resemble.js/）是一个比较引擎,它可以对比两张图片，并找出是否有像素级别的差异。

我们也想使整个测试过程自动化，因此把 PhantomCSS 集成到 Grunt（http://gruntjs.com/）中，然后设置一些自定义的 Grunt 命令，用来运行全部或者部分测试用例。

14.1.2 设置Grunt

在你匆匆忙忙地从 Google 上找到的第一条链接下载 Grunt PhantomCSS 之前，我必须提醒你，你找到的下载链接已经过时了。遗憾的是，有人占据了 GitHub 上 Grunt PhantomCSS 的项目名称以后，就彻底消失了。在这种情况下，一些开发者基于原有的代码库重新创建了自己的 Grunt PhantomCSS 项目，并持续合入新的提交请求，从而保证了 Grunt PhantomCSS 项目的与时俱进。其中一个比较好的实现是由 Anselm Hannemann 维护的（https://github.com/anselmh/grunt-phantomcss）。可以通过如下命令安装：

```
npm i --save-dev git://github.com/anselmh/grunt-phantomcss.git
```

Grunt PhantomCSS 安装成功后，我们需要一些典型的 Grunt 代码，比如在 Gruntfile.js 中加载一个任务。

```
grunt.loadNpmTasks('grunt-phantomcss');
```

Gruntfile.js 中还要设置一些 PhantomCSS 选项，大部分保持默认值就行了。

```
phantomcss: {
  options: {
    mismatchTolerance: 0.05,
    screenshots: 'baselines',
    results: 'results',
    viewportSize: [1280, 800],
    },
    src: [
       'phantomcss.js'
    ]
},
```

- `mismatchTolerance`: 我们可以设置一个视觉误差的阈值，从而可以解释抗锯齿（antialiasing）或者其他因素导致的非关键差异。
- `screenshots`: 用于选择基准图像的存储目录。
- `results`: 运行完对比测试后，结果将存放在这个目录下。
- `viewportSize`: 用 Casper.js 设置视端窗口大小。
- `src`: 用于存放与 gruntfile 文件相关的测试文件的路径。

14.1.3 测试文件

下一步是在 phantomcss.js 文件中设置 Casper.js。PhantomCSS 会启动 PhantomJS 浏览器，而 Casper.js 则用来导航到具体的网页并执行各种所需要的动作。我们认为测试组件的最佳环境是在样式文档中。它和我们的线上站点共享同样的 CSS，而且它也是静态目标地址，我们可以确定它不会一天一变样。因此，我们从使用 Casper 导航到样式文档地址开始：

```
casper.start('http://localhost:9001/cta-link.html')
.then(function() {
  phantomcss.screenshot('.cta-link', 'cta-link');
})
.then(function() {
  this.mouse.move('.cta-button');
  phantomcss.screenshot('.cta-link', 'cta-link-hover');
});
```

通过 Casper 导航到正确的页面后，我们使用 JavaScript 函数链来完成一系列所需的截图。首先选中 `.cta-link` 并进行截图，如图 14-1 所示。我们巧妙地称它为 `cta-link`。这将是它在基准文件夹里的基本文件名。



图 14-1：基准库中的 `cta-link` 图像

然后，我们需要测试按钮，确保当鼠标悬停其上时它的表现符合预期。我们可以使用 CasperJS 在 PhantomJS 加载的页面中移动鼠标，这样我们在抓取下一张截图时，按钮可以处于悬停状态，截图命名为 `cta-link-hover`。图 14-2 展示了结果。



图 14-2：基准库中 `cta-link` 悬停时的图片

14.1.4　对比

有了这些基准图片，我们就可以多次运行测试用例了。如果没有改动，那么后续测试过程中产生的图片将会和基准库中对应的图片完全一致，所有的测试用例都会通过。但是如果有改动的话：

```
.cta-link {
  text-transform: lowercase;
}
```

下次运行对比测试时，我们将会看到图 14-3 中的结果。

正如所料，将大写字母改为小写字母后会导致测试失败。不仅仅是文本有差异，而且按钮也变小了。第三张测试不通过的图片，用粉色标出了两种图片之间的像素差异。[2]

注 2：图 14-3 的彩色图片可从本书页面（http://www.ituring.com.cn/book/1946）的“随书下载”部分获取。
——编者注



图 14-3：一个测试失败的例子

14.1.5　运行全部测试用例

在对每个想要测试的组件（或功能）进行单独的测试后，我们就能运行 `$grunt phantomcss` 了，它会进行如下操作。

(1) 启动一个 PhantomJS 浏览器。
(2) 使用 CasperJS 指向样式规范中的某一页。
(3) 对该页面上的单个单独组件进行截图。
(4) 与页面进行交互：点击 mobile 导航、悬停在链接上、填充表单、提交表单等。
(5) 对以上每一个状态进行截图。
(6) 将这些截图与组件的基准截图（见图 14-4）进行对比。

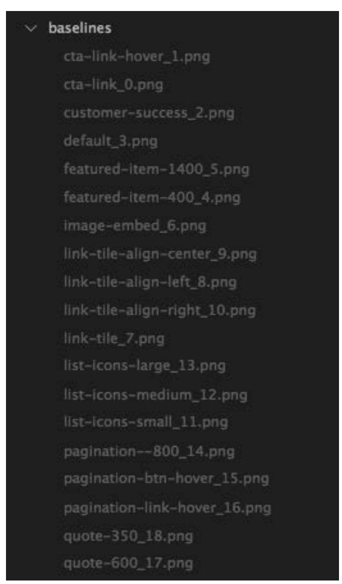


图 14-4：视觉还原测试过程中创建的所有图片

(7) 如果所有图片都与基准截图一致，那么测试通过（PASS!）；如果有改动，那么测试不通过（FAIL!）。

(8) 对库里的每一个组件和布局都重复上述过程。

14.1.6 如何应对测试失败

显然，如果你的任务是修改一个组件的样子，那么你肯定不会通过测试，这是正常的（见图 14-5）。关键在于，你需要保证你只是在你修改的组件上没有通过测试。如果你要修改的是 `cta-link`，而你在 `cta-link` 和翻页组件上都没有通过测试，可能是出现了如下问题。

```
Failure! Saved to /Users/mgodbolt/Sites/webux/baselines/cta-link-hover_1.fail.png
Visual change found for cta-link-hover_1.png (47.62% mismatch)
Failure! Saved to /Users/mgodbolt/Sites/webux/baselines/cta-link_0.fail.png
Visual change found for cta-link_0.png (47.45% mismatch)
Failure! Saved to /Users/mgodbolt/Sites/webux/baselines/pagination--800_14.fail.png
Visual change found for pagination--800_14.png (26.92% mismatch)
Failure! Saved to /Users/mgodbolt/Sites/webux/baselines/pagination-btn-hover_15.fail.png
Visual change found for pagination-btn-hover_15.png (42.57% mismatch)
```

图 14-5：跟踪后续测试的失败报告

- 你改了不该改动的地方。也许你的改动影响范围太大，或者你无意中修改了其他文件中的按钮。不管是哪种情况，找到翻页组件被改动的地方，然后修复它。
- 另一方面，也许你对 `cta-link` 做出的改动注定会对翻页组件有影响。也许它们共享了按钮功能的混入，而且为了保证品牌的一致性，它们应该使用了同样的按钮样式。这时，你应该回过头去找这个功能的创建人、设计师或者这些改动的决策者，问一问是不是这两个组件本身就应该同时修改，然后再来决定。

14.1.7 从失败到成功

不管翻页组件有没有发生变化，`cta-link` 组件还是无法通过测试，因为旧的基准图片已经不正确了。在这种情况下，你应该删除旧的基准图片，并提交新的基准图片（见图 14-6）。如果这些外观上新的变化和品牌视觉规范相一致，那么新的基准图片需要和你的功能分支代码一起提交。这样，当别人合并了你的代码时，这个改动才不会使导致他的测试不能通过。

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   baselines/cta-link-hover_1.png
        modified:   baselines/cta-link_0.png
        modified:   baselines/pagination--800_14.png
        modified:   baselines/pagination-btn-hover_15.png
```

图 14-6：当代码改变了组件的外观时，需要提交新的基准图片

这种方案的优势在于，每一个组件的外观无论何时都有一个绝对的标准，可以用于和当前组件的样子进行对比。你在任何时候的任何分支上都可以运行测试集，而且所有测试都应该能顺利通过，而这正是我们构建系统的坚实基础。

14.1.8 修改代码以适应需求

在 Red Hat 项目中，我首先使用的是 Anselm 的代码，而且发现这些代码能够满足我们 90% 的需求，但剩余的 10% 正是整合工作流所必需的。因此，正如任何有自尊的工程师所做的那样，我从 Anselm 的代码库复制了一个分支，并开始做一些修改，使代码能够满足具体的实现。下面就来看一看我对 Node 模块 @micahgodbolt/grunt-phantomcss（https://www.npmjs.com/package/@micahgodbolt/grunt-phantomcss）所做的一些修改：

```
// Gruntfile.js
phantomcss: {
  webrh: {
    options: {
      mismatchTolerance: 0.05,
      screenshots: 'baselines',
      results: 'results',
      viewportSize: [1280, 800],
    },
    src: [
       // 选择所有以-tests.js结尾的文件
       'src/library/**/*-tests.js'
    ]
  },
},
```

14.1.9 将基准图片放在组件目录里

良好的封装对我们来说很重要，我们将和组件有关的一切东西都放在组件目录里（注意，是一切！）。例如：

- Twig 模板和组件的标准 html 标记
- 描述模板有效数据的 JSON 数据模式
- 组件说明文件，用于解析组件的功能、选项以及实现细节，并填充在 Hologram 样式文档里
- 用于排版和布局的 Sass 文件
- 用于测试组件的每一个变体的测试文件

因为上述组件及相关文件都已经放在了同一个地方，所以我们也希望组件的基准图片同样也放在组件的目录里，这样就能够更方便地找到每个组件的基准图片。而且当一个合并请求包含由于代码改动而产生的新的基准图片时，也应该放在该组件目录下。

PhantomCSS 的默认行为是将所有基准图片放在同一目录下，不管测试文件是在哪个目录。

如果系统中有几十个组件，每一个组件又有多个测试文件，这种方法显然不具有伸缩性。因此我做的关键改动之一就是为每个测试文件新建一个 baseline 文件夹，并将基准图片放入其中（如图 14-7 所示）。

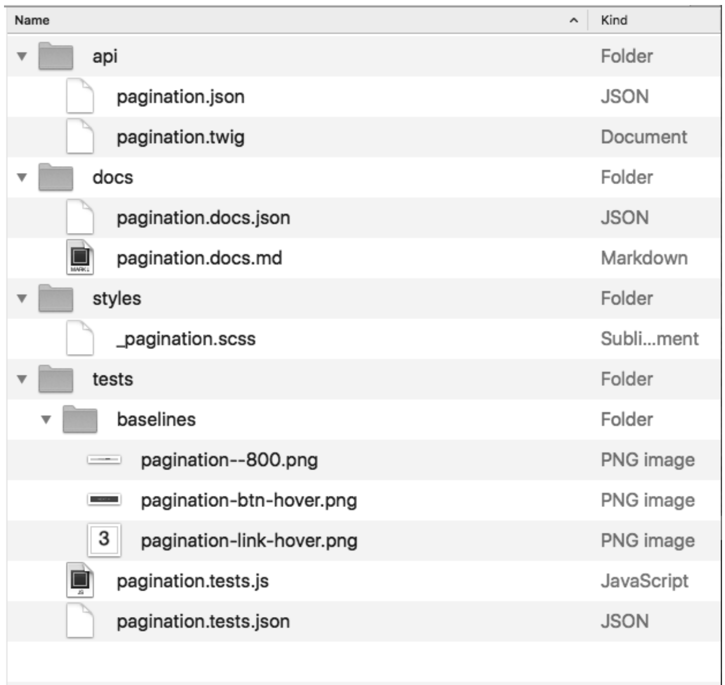


图 14-7：一个典型的组件目录

14.1.10　独立运行每个组件的测试集

除了修改基准图片的路径，我还对组件的测试行为进行了修改，可以将每个组件的测试分别运行，使它们相互独立而不是耦合在一起。因此，现在可以独立地测试每一个组件，看测试是否通过，而不是对 100 多个组件全部测试之后才能知道测试结果，如图 14-8 和图 14-9 所示。

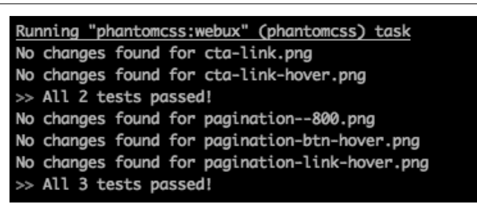


图 14-8：通过所有测试

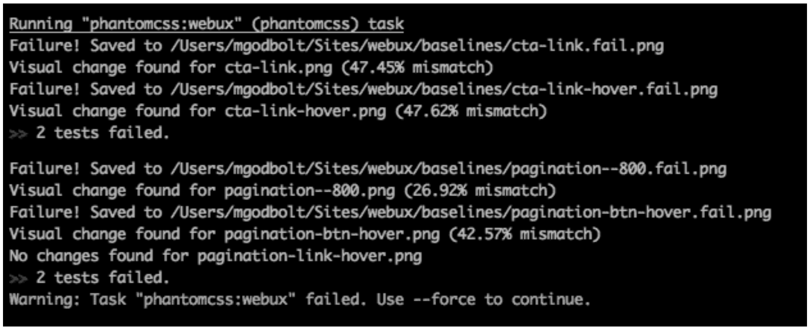


图 14-9：测试失败

14.1.11　测试的可扩展性

我所做的最后一个修改是提高测试用例的灵活性。我把一个测试文件拆分为几十个测试文件，并由 Grunt 运行引入。

测试修改前，原本的实现首先要通过 `casper.start('http://mysite.com/page1')` 启动第一个测试，然后通过 `casper.thenOpen('http://mysite.com/page2')` 启动其他后续的测试。这种方法是有问题的，因为 Grunt 是按字母表的顺序运行这些文件的。于是，只要我新添加的测试文件名中的首字母排在当前测试文件名的首字母之前，测试集就无法运行了。

修改的方法是，在 Grunt 初始化任务时就调用 `casper.start`，这样其余的测试用例就可以使用 `casper.thenOpen` 来启动了：

```
// cta.tests.js
casper.thenOpen('http://localhost:9001/cta.html')
    .then(function () {
        this.viewport(600, 1000);
        phantomcss.screenshot('.rh-cta-link', 'cta-link');
    })
    .then(function () {
        this.mouse.move(".rh-cta-link");
        phantomcss.screenshot('.rh-cta-link', 'cta-link-hover');
    });

// quote.tests.js
casper.thenOpen('http://localhost:9001/quote')
    .then(function () {
        this.viewport(600, 1000);
        phantomcss.screenshot('.rh-quote', 'quote-600');
    })
    .then(function () {
        this.viewport(350, 1000);
        phantomcss.screenshot('.rh-quote', 'quote-350');
    });
```

14.2 小结

将测试用例准备就绪后，就可以很自信地扩展设计系统，添加新的组件、布局和模型了。对每一行新添加的代码，我们都有一套测试集，确保以前的工作不受影响。测试覆盖的不只是组件的基本外观，还包括所有可能的变体或交互时产生的变化。当添加新功能时，我们可以同时编写相应的新测试用例。如果仍然不慎出错，我们可以在修复它之后写一个测试用例，确保它不会再次发生。

如此一来，当我们给设计系统添加新功能时，会发现管理起来越来越容易，而不是越来越困难。现在我们可以持续优化和改进当前的组件，不用担心破坏其他的组件了。

第五部分

文档核心

面对这一事实吧：我们启动的每个项目的前端代码都正在变得越来越复杂。我并不是说这是件坏事，只是快速的成长也带来了更多的问题。

就在几年前，我们还把所有的 CSS 放在一个单一的文件中，并且对每一个样式使用一长串很复杂的选择器，以此来确定页面中对应的元素。如果发现某一个样式受到页面上其他样式的干扰，就在文件底部写一个有更长的选择器的新样式。

与之类似，我们的 JavaScript 文件曾经也使用一大堆 JQuery 函数，以此对页面中的目标元素应用某些功能。每个函数都包含了完成所需功能的全部逻辑和代码，如果需要对另外一个元素实现稍微不同的功能，只要简单地复制一份代码，修改一下选择器，并更新部分逻辑即可。

简而言之，我们过去相当于为每个项目写一个单独的 PHP 文件应用程序。

之后，正如 PHP 开发者学着将代码拆分成可重用的对象，并将代码组织到不同的文件中一样，前端项目也逐渐摆脱原先一长串级联指令的形态，开始变得更像一个充满抽象定义、依赖关系和接口的复杂系统。虽然我们很快采用了传统编程语言中的面向对象和多文件的方法，但是很难做到同步写出完整的文档。

这么多年来我们一直使用声明式系统，以至于我们对新系统的理解在头脑中早已固化了。将头脑中这些看似常识的信息写到文档上并非易事，但我们因为没有文档而浪费的时间比写文档花费的时间更多。俗话说得好：好记性不如烂笔头。

何为文档

文档是系统设计的蓝图。没有文档，我们将难免重复解决已经解决过的问题，而且花大量时间查看代码来寻找最简单的答案。没有文档，我们的新员工只能对着系统抓耳挠腮，并怀疑在这种系统中怎么可能完成自己的任务。

回顾一下目前为止我们设计过的所有架构，如果不花同等的时间来讲一下写文档的方法，那简直是一种罪过。写文档是开发工作的一部分，而不是等重要工作完成后才开始的事情。就像需要重构的臃肿代码、需要自动化的低效率流程，或者没有被测试覆盖到的函数一样，略过文档也会欠下技术债。

不要以为文档只是简单地写下代码如何工作。的确，我们需要在开发流程中预留出写文档的时间，用于记录我们开发的代码是如何工作的，但是写文档远远不止为每一行代码写一段描述。

文档有多种形式，而其中很多只有在架构支持时才能成型。虽然有些文档只是用于描述每个函数的普通文本，但这种文档背后往往有一套基于搜索、导航和视觉呈现的构建系统。其他的文档用于展示系统的资源，由我们所写的样式、脚本、模板和模式来驱动。

静态文档

Hologram（https://github.com/trulia/hologram）是基于 Ruby 的通用文档工具，你可以在代码库中写小段的注释，然后通过它来收集这些注释生成的静态页面。 这些 Markdown 格式的文档块可以放进你的 Sass、CSS 或者 JavaScript 文件中。 这些文档块还包括用于描述页面名称和导航等相关信息的元数据，并且它的书写形式完全自由。Hologram 让你可以将文档内联地写在代码中，这有助于使文档保持最新，同时使开发人员总能看到这些文档。

SassDoc（http://sassdoc.com/）是基于 Node 的系统文档工具，它宣称“SassDoc 对于 Sass 的意义，就像 JSDoc 对于 JavaScript 的意义一样”，而且它的确如此！ SassDoc 与 Hologram 有点像，它也依赖于代码中内联的注释来生成最终文档。然而，Hologram 是通用的、多功能的工具，SaasDoc 却专注于描述 Sass 变量、函数、混入（mixin），以及它们是如何相互影响和依赖的。如果你正在构建一个大型的 Sass 框架，或者一个复杂的栅格或颜色系统，SassDoc 正是你想要的工具。

代码驱动的文档

Pattern Lab 是多平台模式库工具，它使你可以模块化地开发设计系统，并将模板和 CSS 转换成可浏览的模式库。在模块化的系统中，你可以先开发每个单独的模式片段，然后通过组合这些片段产生更复杂的模式。Pattern Lab 提供了一个基本框架，用于模块化地创建并组合这些片段，使之成为更复杂的模式，甚至还能输出完整的页面。可预览的组件库是开

发者、设计师、用户体验师、质量工程师和产品所有者聚在一起时可以使用的完美工具。它为设计系统中的每个部分创建了一门通用的语言和稳定的参照系。

JSON 模式是用于描述数据格式的语言，同时也可以说明数据的验证方式。在前端架构的领域中，可以用 JSON 模式来描述模板和模式所需要的数据。JSON 超模式甚至可以描述能够通过 HTTP 协议与设计系统交互的方法，包括验证、渲染和测试。JSON 模式是一种代码驱动的文档工具，因为它提供了验证和驱动编辑工具的功能。JSON 模式还提供了可读性很强的系统手册，取代了开发者实现一个功能所需的一大堆手写说明。

第 15 章

样式文档

既然样式表已从一长串的声明演化为一个拥有变量、函数和逻辑的系统，那么也要保证文档系统的演化能跟上节奏。Hologram（http://trulia.github.io/hologram/）提供了构建稳健的文档系统所需的一切内容。它允许我们在实际编写代码时，直接在 Sass 或者 JavaScript 文件中给系统进行注释。Hologram 会自动收集这些注释的内容，并转化为一个含有渲染示例和代码示例的可访问网站。这意味着我们不是在维护两个单独的代码库，而是将样式文档和实际的设计系统结合在一起。如果有任何代码的注释不够充分，那么它就会很容易被发现。

在系统文档上，Hologram 允许我们在样式文档中创建可访问的标准 Markdown 文件。这给了我们一个绝佳的位置去编写入门文档、项目规则和章程、联系方式，以及其他任何需要收集整理并展示给团队的信息。

下面快速了解一下 Hologram 的配置，然后再看一些文档的例子。

Hologram 是一个 Ruby 库，因此我们从安装库开始：

```
$ gem install hologram
```

15.1 配置Hologram

安装库之后，我们创建一个 YAML 文件来配置 Hologram：

```
destination: ./docs
documentation_assets: ./doc_assets
code_example_templates: ./code_example_templates
dependencies: ./build
source: ./sass
```

接下来仔细看代码的每一部分。

`destination`

这是 Hologram 用于输出静态样式文档的文件夹。如果把它设置为可通过外部访问，那么它将会成为公共文件夹，并保存一切样式文档所需的内容。

`documentation_assets`

这是用来搭建样式文档的静态资源文件。我们需要指定用于生成 HTML 的模板，还有用于改进用户体验的任何 CSS 或 JavaScript。这包括如代码高亮、示例布局、导航样式等内容。如果你还不想为了给网站写文档而去搭建另一个网站，那么可以在 Hologram 的 GitHub 页面（https://github.com/trulia/hologram#extensions-and-plugins）底部所列的现成模板中挑选一个使用。

`code_example_templates`

代码示例和它们在浏览器中的渲染效果是本样式文档的核心。Hologram 提供了一种非常便捷的方式来自定义示例周围的标记。下面是 Hologram 使用的默认标记，可以根据自己的需要随意修改：

```
<div class="codeExample">
 <div class="exampleOutput">
   <%= rendered_example %>
 </div>
 <div class="codeBlock">
   <div class="highlight">
     <pre><%= code_example %></pre>
   </div>
 </div>
</div>
```

请注意 `rendered_example`（渲染示例）和 `code_example`（代码示例）两个变量。渲染示例是这个例子在页面中直接显示的效果，而代码示例被放置在 `<pre>` 标签当中，不对其进行渲染，并且加上了代码高亮的类名。图 15-1 展示了一组代码示例的渲染效果。

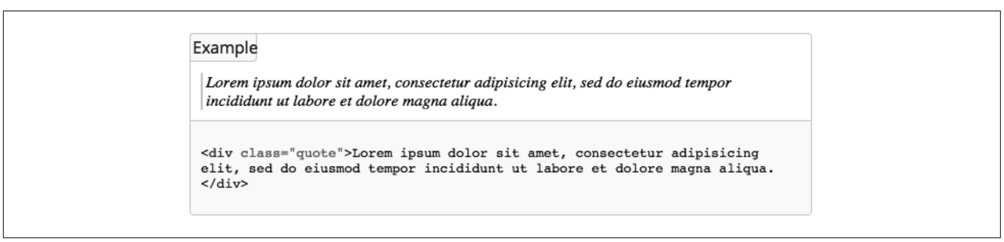


图 15-1：一个引用组件在 Hologram 样式文档中的渲染效果

dependencies

与 documentation_assets 相似，dependencies 描述了一系列样式文档文件夹中需要包括的资源。这些路径中包含编译后的 CSS、JavaScript、字体文件、图片文件，以及其他任何用于正确展示和与样式文档中的内容进行交互的资源。

source

最后需要告诉 Hologram 我们希望在样式文档中出现的文档放在哪里。这一系列的源文件可以也应当包括 Sass/CSS、JavaScript/CoffeeScript、HTML 模板、图标字体文件夹，以及保存入门手册、工作流或测试程序等内容的文件夹。下面来看一个文档示例，帮助理解各个值的含义及其在样式文档中该如何显示。

15.1.1 Hologram的文档注释块

下面这个例子可能出现在你的 _buttons.scss 文件中。Hologram 的内容被放置在一个 CSS 注释中，以关键字“doc”开头：

````
/*doc
---
title: Primary Button
category: Base CSS
---

This is our button

```html_example
<a class="btn" href="#">Click</a>
```
*/


.btn {
 color: white;
 background: blue;
 padding: 10px;
 border: none;
 text-decoration: none;
}
````

接下来看看代码中的每个部分。

作为 Markdown 文件，它允许我们使用代码块，不管是单行还是多行。Hologram 带有一个自定义的 Markdown 渲染器，采用一些代码块的关键字（html_example、js_example、haml_example，等等），这些关键字不仅会使标记带语法高亮，而且还会把标记渲染到页面中。

图 15-2 显示了这个按钮在 Hologram 样式文档中的渲染效果。



图 15-2：一个基础按钮在样式文档中的渲染效果

`title`

　　这是为这篇文档定义的可读性较高的标题。

`category`

　　样式文档中的每一个分类（category）都有自己单独的页面，并且在导航中有其对应的条目。你可以在每个分类中添加任意数量的条目。默认模板会提供跳转到每个文档标题的链接。

`Documentation body`

　　在 ``` （结束符号）之后，就是文档的主体部分。这个区域作为 Markdown 来解析，因此我们不需要任何 HTML 标签，就可以轻松引入标题、列表、链接、图片，甚至还有表格。不过如果我们希望使用 HTML，当然也是没有问题的。

15.1.2　Hologram编译流程

Hologram 运行时，会遍历源目录中的所有文件，提取所有被 `/*doc */` 包围的内容到样式文档中，然后使用 YAML frontmatter 来定义它们的标题和分类。最终生成的是一个包含所有依赖资源（CSS、JavaScript、图片）和每个分类所对应的 HTML 文件的文件夹。

每个页面都包括对应分类下的所有文档，这些标准 HTML 页面都是由 Markdown 和代码块转化而来的。然后，这些页面被加入头部和底部模板中，再补充上 CSS、JavaScript、字体和导航元素。你可以自定义核心内容的模板，而如果你将在这个样式文档上花费大量时间的话，你可能要重写大部分内容，并加入你自己的品牌和样式。

把文档内容写在注释里面是一种非常好的做法，因为这样可以让文档紧挨在对应的代码的后面，而且还不会影响代码对我们的设计系统风格的实际作用。CSS 注释很容易在模块的 CSS 中移动，在你需要发布代码时也很容易删除。

15.1.3 Hologram小结

现在可以在项目的源文件中编写大段的 Markdown 文档了，Hologram 会收集它们，并且把它们转化为一个小型网站，我们还可以对网站进行个性化定制、添加搜索或过滤等功能，或者设计成符合企业品牌的风格。文档甚至无需跟代码处于同一个文件中，我们可以把文档分离出来，作为单独的文件放在代码后面。每一个组件都有单独的文档文件，它们和样式放在同一个文件夹中。另外还有与 Sass 或者 JavaScript 完全无关的文档，它们只记录项目配置和工作流的信息。当时，我们很快就发现了 Hologram 文档的多种用法，相信你也会跟我们一样觉得它非常好用。

15.2 SassDoc

Hologram 是一个格式非常自由的文档系统，它并不关心它的内容，以及它写在什么类型的文件中。它就像是一张白纸，它是什么取决于你让它成为什么。

SassDoc（http://sassdoc.com/）则完全相反。它是一个记录 Sass 的变量、混入、继承和函数的工具，而且对每个注释块后面的代码都有严格的要求。另外，它对文档的呈现和风格也很讲究。Hologram 只是把文档分成不同的组，而 SassDoc 则会自动建立起变量和函数或者混入和扩展的映射关系。如果你在编写一个大型的基于 Sass 的设计系统或者一个 Sass 框架，而且你不想手写所有的文档，那么 SassDoc 将是你的最佳选择！接下来看看如何开始使用 SassDoc。

15.2.1 安装SassDoc

SassDoc 是一个基于 NodeJS 的文档系统，而且拥有 Grunt、Gulp 和 Broccoli 的插件。你可以在命令行中对你项目中的 Sass 文件运行 SassDoc。鉴于输出的内容总是一致的，为了简单起见，我将演示命令行选项。这种方式可以让你在不改动项目的前提下开始使用 SassDoc。

我们从把 SassDoc 安装为全局的 NPM 包开始（再次提醒，这只是为了能更简单地试用 SassDoc，如果在实际项目中使用，你可能不会使用全局选项）：

```
$ npm install sassdoc --global
```

把 SassDoc 安装进全局 NPM 文件夹后，就可以直接在 sass 文件夹中运行 `sassdoc` 指令了：

```
sassdoc sass
```

这就是它的用法！ SassDoc 会遍历 sass 文件夹中的所有文件，并提取出所有符合 SassDoc 文档特殊语法的内容。然后，它会使用这些信息构建一个功能齐全的静态文档网站。接下来看一下 SassDoc 的语法，以及带有 SassDoc 格式注释的代码在编译后会变成什么样。

15.2.2　使用SassDoc

SassDoc 使用三个斜杠的 Sass 注释来指定哪些代码是需要被纳入到文档中的。你需要做的就是把 /// 放在一个混入、扩展、函数或者变量的上面，然后 SassDoc 就会帮你做其余的事情。下面是一个仅仅用到两个变量的例子：

```
///
$button-padding: 1em;
$button-font-size: 1.2em;
```

在前面的例子中，变量 `$button-padding` 会被导入 SassDoc，而变量 `$button-font-size` 则不会。SassDoc 能够智能地抓取紧跟在标记后面的变量，即使你把两个变量放在同一行中，SassDoc 仍然只会导入第一个。如果你希望两个都导入，可以这样写：

```
///
$button-padding: 1em;

///
$button-margin: 1.2em;
```

仅仅用了两个变量、六个斜杠和一个终端命令，SassDoc 就创建了如图 15-3 所示的文档。

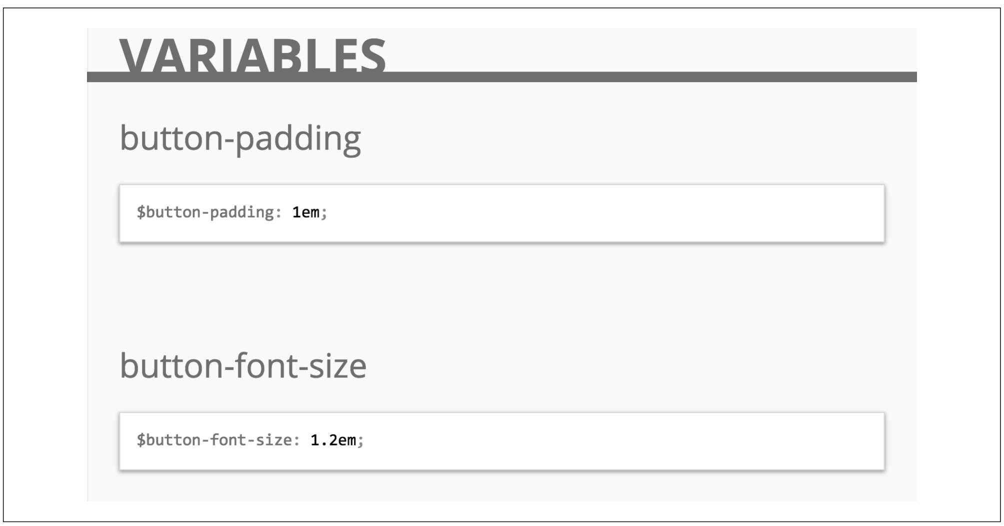


图 15-3：渲染后的 SassDoc 样式文档

哇！我们不需要创建任何模板文件或者编写任何 CSS 代码，就拥有了看起来非常专业的，由静态 HTML、CSS 和 JavaScript 构建而成的文档页面。简单地把代码推送到 GitHub 之后，你就拥有了可以展示你的系统的一系列变量的文档。不过，SassDoc 可以做的事情远不止这些，这才刚刚开始。

15.2.3 探索SassDoc

现在来创建另一个变量，并看看 SassDoc 的一些注释类型：

```
/// A number between 0 and 360 used to find __foreground color__
/// @type Number
/// @access private
$foreground-adjust: 180 !global;
```

这里的变量前面有几行以三个斜杠开头的 SassDoc 注释，它们都会随着变量名和变量值被一起纳入到文档中。

注释块中的第一行往往都是 SassDoc 用于展示在变量上方的描述。描述可以有多行，而且是作为 Markdown 格式来解析的，因此它支持标题、列表、链接和其他任何你想加入的内容，只要保证每一行的开始都带三个斜杠即可。

在描述之后，我们就可以自由地使用 SassDoc 众多的注释类型中的任何一个了。为首的是 `@type` 注释。如果这个变量仅仅是另外一个变量的引用，或者是一个函数的返回值，那么 `@type` 注释有助于说明这个变量的类型是字符串、数字、布尔值，还是映射。实际上你可以使用任何你喜欢的文本字符串来描述 `@type`，因此请写下对读者最有帮助的内容。

这里使用的另一个注释是 `@access`。这个注释有两个可能的值：`private` 和 `public`。如果你有传统的面向对象语言的编程背景，那么你对变量和函数的 `private` 属性一定很熟悉，这代表它们只能在对象内部被访问，而不能被系统的其他部分所调用。Sass 实际上并没有这类技术限制，但如果创建的是只在内部使用的变量、函数、混入或扩展，那就应该把 `@access` 设置为 `private`。这对你的系统而言并没有功能上的改动，但是它的确告诉用户不要在他们的样式表中使用这个变量，因为它随时可能被改动、移除或重构。

如图 15-4 所示，`@access private` 在变量的标题前面加上了 `[Private]` 标记，描述按照 Markdown 语法全部被解析出来，并且在变量下方指明了变量的类型。

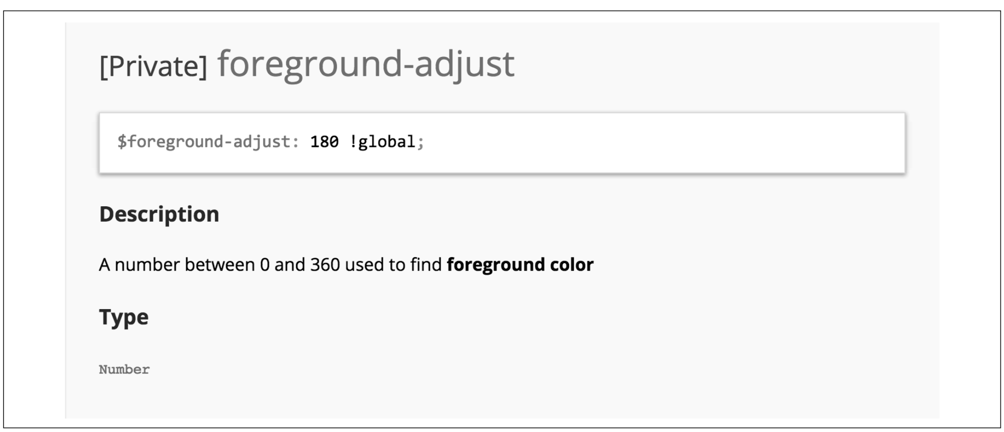


图 15-4：变量 `$foreground-adjust` 在 SassDoc 中的渲染效果

15.2.4　深入了解SassDoc

既然已经见识了 Sass 设计系统文档的基本用法，接下来就看一下一套更完整的变量、函数、混入和扩展的样子。下面是一个稍加构造的按钮混入，以及输出按钮 CSS 所需的一切内容。

```
/// Our global button padding
/// @access private
$button-padding: 1em !global;

/// Our global button font-size
/// @access private
$button-font-size: 1.2em !global;

/// A number between 0 and 360
/// @type number
/// @access private
$foreground-adjust: 180 !global;

/// Function to return a foreground color based
on a background color
/// @access private
/// @param {color} $color - The background color
/// @return {color}
/// @example
///   @function get_foreground(blue);
///   // yellow
@function get_foreground($color) {
 @return adjust_hue($color, $foreground-adjust);
}

/// Our core button styles
/// @access private
%btn-core {
 padding: $button-padding;
 text-decoration: none;
 font-size: $button-font-size;
}

/// Our basic button mixin
/// @access public
/// @param {Color} $bg_color [red] - Background Color
@mixin button($bg_color:"red") {
 background: $bg_color;
 color: get_foreground($bg_color);
 @extend %btn-core;
}
```

在这些变量、函数和扩展当中，`@mixin button` 是系统中唯一对外公开的部分。这保证用户只会调用这个混入，而永远不会使用 `@extend %btn-core`、`get_foreground` 或者任何变量。当我们决定重构这个按钮混入时，我们可以移除或改变这些私有元素中的任意一个，而不用担心破坏代码。

我们的按钮混入使用了 `@param` 注释，它让我们可以记录混入所用到的各种输入参数，然后用一种清晰且统一的格式展示出来。`@param` 的书写格式如下：

```
@param {type} $param_name [default value] - description
```

15.2.5　内部依赖

SassDoc 最强大的功能之一就是可以自动列出内部的依赖关系。如果没有静默的 `%btn-core` 扩展，我们的按钮混入就会失效。实际上按钮混入还依赖 `get_foreground` 函数，而 SassDoc 可以追踪所有的依赖关系，并把它们展示在混入的文档底部。

综合所有的内容，图 15-5 展示了这个按钮文档最终的样子。

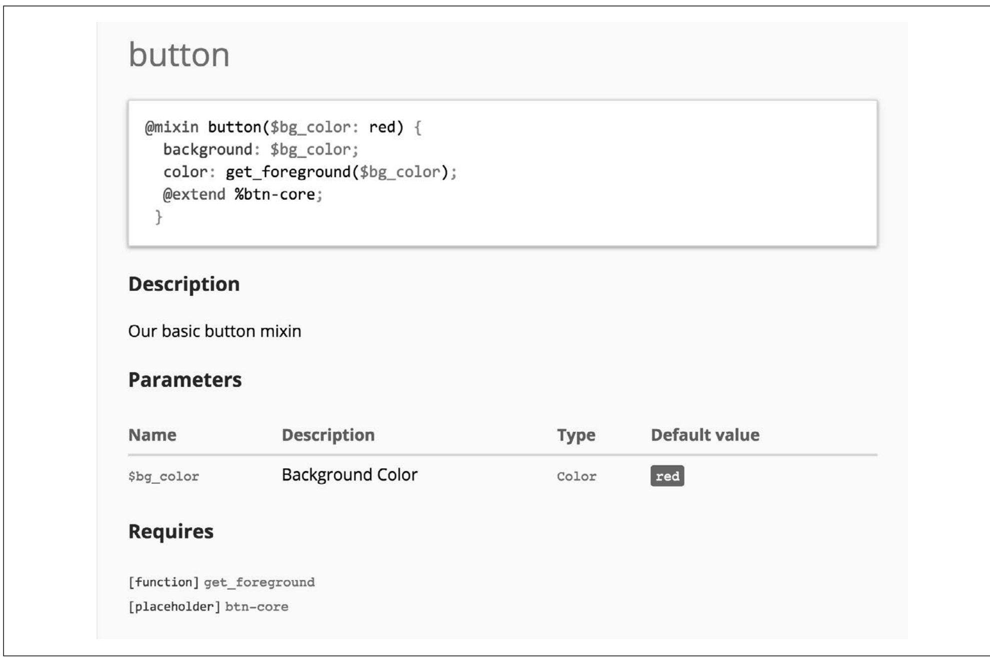


图 15-5：按钮混入在 SassDoc 中的展示效果

`%btn-core` 占位符并没有使用任何新的注释类型，不过我们的按钮混入依赖于它，而它又依赖于一些变量，因此那些依赖关系就自动地罗列出来了，如图 15-6 所示。

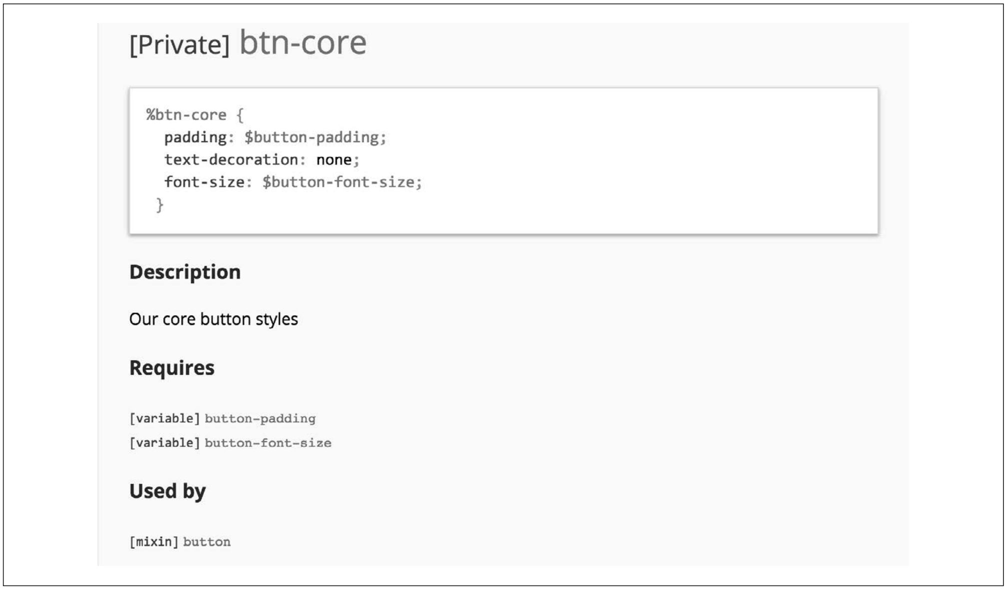


图 15-6：`%btn-core` 扩展在 SassDoc 中的展示效果

如图 15-7 所示，`get_foreground()` 函数展示了 `@return` 注释的一个例子，它允许我们描述期望的函数返回值的类型。这个函数还使用了 `@example` 注释，它允许书写一个缩进的代码块来演示函数的使用方式，以及如果按照例子实现之后会得到什么样的返回值。同时注意“Requires”和“Used by”这两个区域，这些自动生成的依赖关系是非常有价值的，而且它们中的每一个都是跳转到混入或者变量定义的链接，使得文档非常易于浏览。

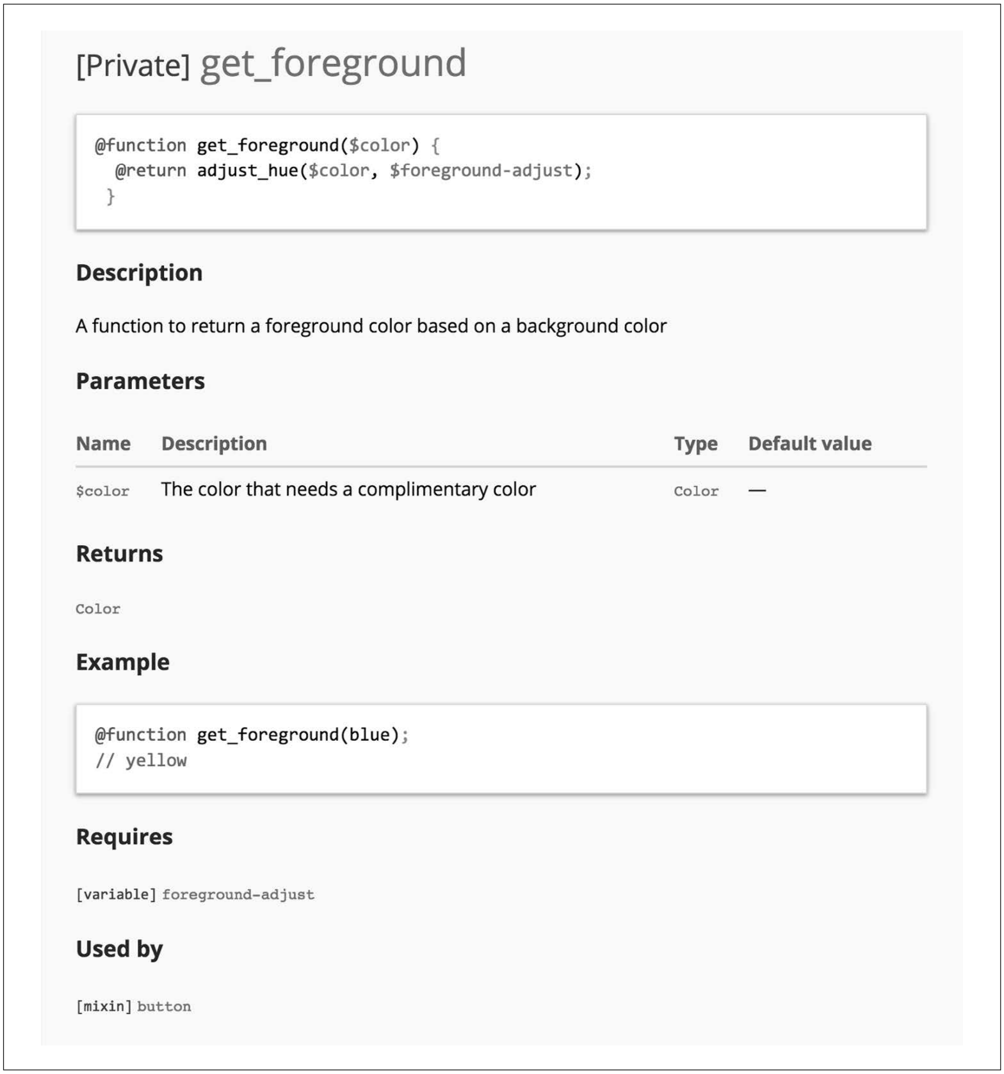


图 15-7：get_foreground 函数在 SassDoc 中的展示效果

15.3　小结

正如本章的示例所阐释的，静态文档系统可以是获取设计系统中的输入和输出的有效途径。Hologram 为创建一系列开放和个性化的文档页面提供了工具。无论是编写入门文档，还是获取一个行动召唤（call to action, CTA）按钮的正确标记，Hologram 都会提供充分的灵活性，让你编写、归类，并且以想要的风格来展示。

如果你在寻找为你的 Sass 系统创建更严格和更标准的视图的文档工具，那么 SassDoc 就是

你上手的最佳选择。虽然它仍然是静态的文档工具，但它通过连接系统的依赖关系，提升了构建文档的自动化程度。

事实上，这两个工具不是彼此独立的。很多现代的样式文档是用几种不同的工具构建的，每一种工具尽可能用最佳的方式来展示系统的不同部分。这两个工具的共同点在于，它们都依赖大量的用户输入来发挥真正的潜力。作为静态文档工具，它们提供了一种方式，让我们能够为正在搭建的网站编写文档，并把所有的信息整理、归类并建立索引。

此外，代码驱动的文档系统是另一种方式。在代码驱动的环境下，文档纯粹是由我们所编写的代码生成的。下一章将会举一些相关的例子。

第 16 章

图形库

首先来介绍一下 Brad Frost 的原子设计原则（http://patternlab.io），以防读者之前对此并不了解。原子设计原则无疑会改变你看待网站设计的方式，因此如果有机会的话，最好能够深入研究一下这个领域。

原子设计是一种构建网站设计系统的方法论。它不是一次性地设计整个页面，而是首先把网页常用元素分解成各个尺寸的模式，然后再描述把这些模式组合成一个完整网页的方式。这些模式中不可再分的最小单位就称为原子。

*原子*是构造网站、标题、列表样式、图片、视频和表单元素的基本结构单元。

就像大自然中的原子能组合成更复杂的东西，原子设计把多个原子组合成*分子*。在网页设计中，“分子”可能代表一个搜索表单、媒体块或者导航栏。

正如多个原子组成多个分子，多个分子组成*有机体*来创造界面的一部分，如博客文章模块或评论模块。

最后创造出的是*模板*，它代表同一个页面上的多个界面元素的组合和布局，包括导航栏、内容区域和底部元素。

16.1 何为Pattern Lab

既然原子设计已经提供了一种通过可分解的模块来组合成网站的方法论，如果再有一个工具让我们轻而易举地实践它，把新的网页设计做成原型，那岂不是很令人振奋？而这正是 Pattern Lab 的用武之地。

Pattern Lab（又称 PL）是一个静态网站生成器，它可以记录所有的网站结构单元。它把原子、分子、有机体和布局变成可浏览的网页，你不仅可以从中看到每个原子组件，还可以使用自定义的内容创造样例页面，模仿实际的网站效果。

Pattern Lab入门

Pattern Lab 是一个小型网站，自带很多 CSS 文件、模板、JSON 数据文件，以及一个构建系统（用来创造可浏览的静态页面）。原版本是基于 PHP 的，而我将要演示的是一个 Node 版本。开始使用时，你可以选择直接把 Pattern Lab 库粘贴到你的项目里，也可以使用 Yeoman 这样的工具来快速生成新项目。

使用 Pattern Lab 最好的方式是把它放到当前的系统主题的目录里，或者至少放到能与项目共用文件的地方。这么做是为了确保 Pattern Lab 和你的网站共用同样的 CSS 和 JavaScript 文件。这样你就可以在 Pattern Lab 中设计原型，创造所有必要的样式和脚本。并且当 HTML 在 CMS 系统里生效时，所有必要的 CSS 和 JavaScript 也已经就位了。

当 Pattern Lab 下载完成并安装好所有相关的 Node 模块后（如果你使用的是 Node 版本），用一句命令就可以让你的 PL 网站运行起来。Node 版本的运行命令为 `$ grunt server`。在稍加编译并启动一个简单的 Web 服务器之后，我们会看到一个预置的简单的样式文档，有原子、分子、有机体、模板，甚至还有完整页面的示例，就像图 16-1 所展示的主页。

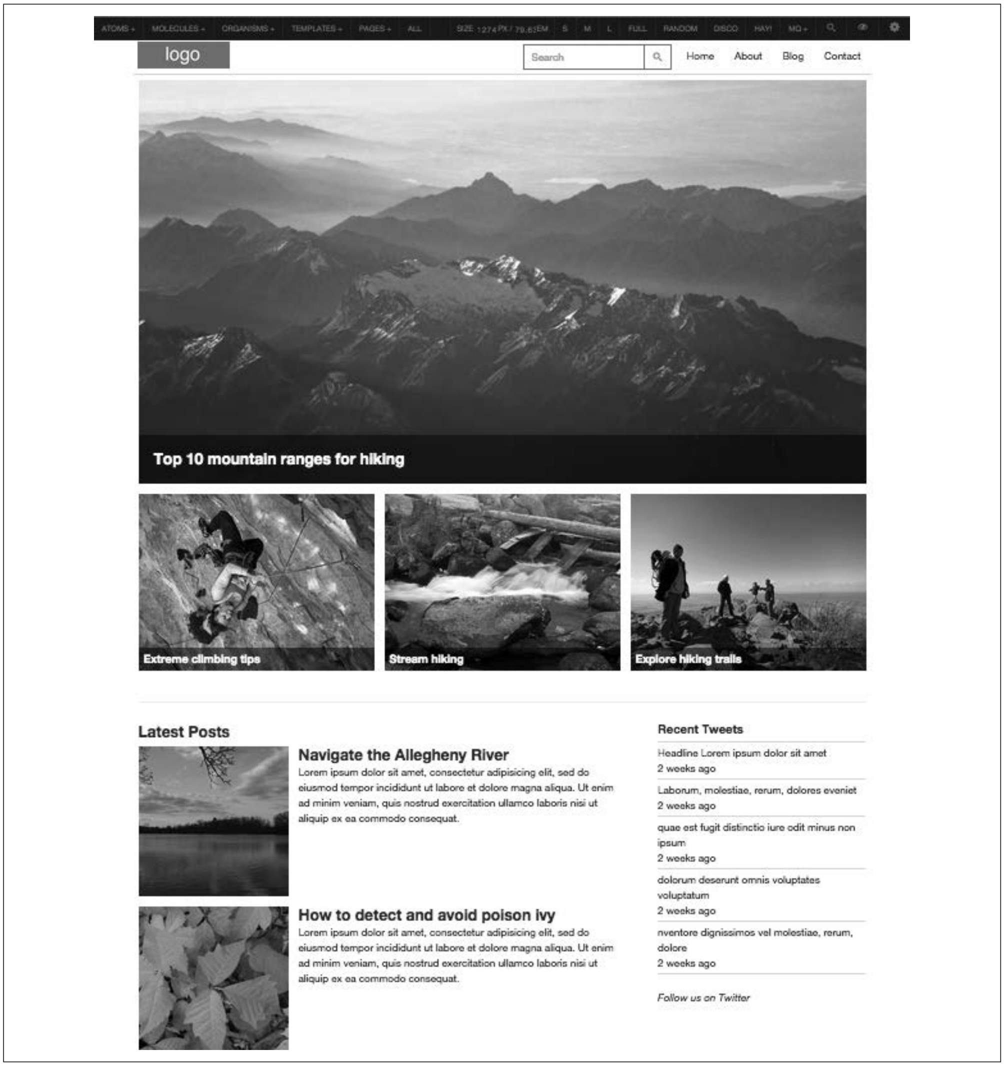


图 16-1：Pattern Lab 里的首页预览

16.2　运行Pattern Lab

理解系统背后的运行原理的最好方式，就是从一个完善的界面开始，逆向推出系统的模板，还有 Pattern Lab 用到的数据文件。因此，下面就从广告图（那座山）开始，看看如何才能在页面上展示这种带标题的图片。

首先要明白，所有的文件按不同层级，分别放在 5 个不同的文件夹中：原子、分子、有机体、模板、页面。每个模式都有一个文件，描述组成按钮、轮播图片或底部导航栏的标

记。在页面这一层级上，只需选择模板并添加内容就可以生成页面，上面的模式也会有关联文件，让我们可以把 lorem ipsum 字符（或“乱数假文”）替换成真正的内容。接下来看主页模式的范例，在一番研究之后，我们会发现雪山广告图有这样一个图片标签：

```
<!-- 00-homepage.mustache -->

{{> templates-homepage }}
```

正如你所见，Pattern Lab 使用 Mustache 格式的文件，Mustache 是一种基本的模板语言，它支持使用变量、通过迭代器遍历数据，以及导入其他文件。在 00-homepage.mustache 文件中，我们只需要导入主页模板，无需其他任何操作。“页面文件”的作用就在于，它可以把一些数据和模板关联在一起，下面就来看看对应的数据文件：

```
<!-- 00-homepage.json -->

{
 "title": "Home Page",
 "bodyClass": "home",
 "hero": [
   {
     "img": {
       "landscape-16x9": {
         "src": "../../images/sample/16x9-mountains.jpg",
         "alt": "Mountains"
       }
     },
     "headline": {
       "medium": "Top 10 mountain ranges for hiking"
     }
   }
 ],
 "touts": [
   ...
 ],
 "latest-posts": [
  ...
 ]
}
```

有了数据填充的视图，我们的页面就有了一个很好的轮廓。可以看到，数据文件里有一个标题和一个 CSS 类名，接下来是广告图区域和宣传文案，然后是最新的帖子。

JSON（http://www.json.org）是一个用于保存数据的极好的格式。其中用双引号（`""`）包含字符串，用方括号（`[]`）包含数组，用花括号（`{}`）包含对象。我们可以依此创建一个字符串数组 `["like","this"]`，甚至还可以创建对象数组，就像刚才的数据文件例子中，文件里每一个段落都是一个对象数组，比如“hero”“touts”等。

既然现在已经有了数据，下面就来进一步了解页面调用的第一个模板文件，即首页模板：

```
<!-- 00-homepage.mustache -->

<div class="page" id="page">
  {{> organisms-header }}
  <div role="main">
    {{# hero }}
      {{> molecules-block-hero }}
    {{/ hero}}
    <div class="g g-3up">
      {{# touts}}
        <div class="gi">
          {{> molecules-inset-block }}
        </div>
      {{/ touts}}
    </div><!--end 3up-->
    <hr />
    <div class="l-two-col">
      <div class="l-main">
        <section class="section latest-posts">
          <h2 class="section-title">Latest Posts</h2>
          <ul class="post-list">
            {{# latest-posts }}
              <li>{{> molecules-media-block }}</li>
            {{/ latest-posts }}
          </ul>
          <a href="#" class="text-btn">View more posts</a>
        </section>
      </div><!--end .l-main-->
      <div class="l-sidebar">
        {{> organisms-recent-tweets }}
      </div><!--end .l-sidebar-->
    </div><!--end .l-two-col-->
  </div><!--End role=main-->
  {{> organisms-footer }}
</div>
```

16.3　首页模板

首先你可能会注意到，这个模板并不是以 template-homepage 命名的。Pattern Lab 采用了智能文件解析系统，因此只需要输入 `<type>-<name>`，无需传递具体的路径，甚至也不需要文件名。于是，当我们需要引入名为 header 的“有机体”时，我们不用关心引入文件的实际路径 _patterns/02-organisms/00-global/00-header.mustache，只需要输入 `{{> organisms-header }}` 即可。

拉取 `organisms-header` 之后，模板文件就导入了广告图页面。导入的每个段落都可以用 `{{# }}` 标签包含起来，表示数据上下文的改变。

```
{{# hero }}
     {{> molecules-block-hero }}
{{/ hero}}
```

这意味着，如果页面中导入了名为 `block-hero` 的分子，那么它会使用 `hero` 数据对象来加载里面的所有变量。

```
{
  "img": {
   "landscape-16x9": {
     "src": "../../images/sample/landscape-16x9-mountains.jpg",
     "alt": "Mountains"
   }
  },
  "headline" : {
   "medium" : "Top 10 mountain ranges for hiking"
  }
}
```

现在我们知道使用的是哪些数据了，下一步来看看 Mustache 文件 block-hero：

```
<div class="block block-hero">
 <a href="{{ url }}" class="inner">
   <div class="b-thumb">
     {{> atoms-landscape-16x9 }}
   </div>
   <div class="b-text">
     <h2 class="headline">{{ headline.medium }}</h2>
   </div>
 </a>
</div>
```

16.4 首变量

这个 Mustache 文件保存了我们的首变量。在 Mustache 语法里，变量就是包含在 `{{}}` 里的词。你可能留意到这里的首变量是 `{{url}}`，它在我们数据集里还没有一个关联的值。如果想要指定这个 URL，就需要给数据集添加一个关键词为 `url` 的属性，然后这个模板就会调用赋给 `url` 属性的值。

```
{
   "url": "http://www.google.com",
   "img": {
       "landscape-16x9": {
           "src": "../../images/sample/16x9-mountains.jpg",
           "alt": "Mountains"
       }
   },
   "headline" : {
       "medium" : "Top 10 mountain ranges for hiking"
   }
}
```

然而我们并没有指定值，这个时候 Pattern Lab 可以智能地调用全局的 data.json 文件里定义

好的 `url` 默认值。

下一个变量是 `{{ headline.medium }}`，这确实是一个我们有的值。中间的点号意思是在 `headline` 对象里找到 `medium` 的属性名，然后返回它的值：`Top 10 mountain ranges for hiking`。

16.5 原子

我们还需要解决最后调用的模板 `{{> atoms-landscape-16x9 }}`。就像我们在首页模板中看到的，这个标签导入了名为“landscape-16x9”的原子，但是这里并没有改变数据的上下文。这意味着这个负责显示广告图的 Mustache 文件会和 `block-hero` 使用完全一样的数据，不过这也不算什么大问题，因为 `block-hero` 模板中几乎没有多个 `img` 属性的需求。因此，可以使用和 `block-hero` 相同的数据调用模板并传入广告图。

```
<!-- 02-landscape-16x9.mustache -->
<img src="{{ img.16x9.src }}" alt="{{ img.16x9.alt }}" />
```

Pattern Lab 让我们可以抽象最简单的标记块，这个图片标签就是一个很好的例子。这个模板只调用了图片源码及其 `alt` 标签，因此看起来可能多此一举。为什么不直接放一个图片标签到广告图板块里，就像那个 H2 标题一样？话说回来，究竟把模板抽象到什么程度，还是应该由你和你的团队来决定，不过所有图片标签都用单一的模板生成是很有意义的。

想象一下将来你决定给所有的图片标签添加一个 CSS 类名或者 data 属性。如果每个模板只处理自己的图片标签，那就需要更新很多“分子”。但是如果所有的图片都用一个单独的模板来处理，那么只需更新一处，Pattern Lab 就会帮你更新其他任何用到这个原子的地方。

16.6 发挥原子的作用

现在，我们已经见识了在首页显示广告图的方法，还可以采取同样的方法分解页面上的每一个 UI 元素。这种方法的强大之处在于，在持续构建的过程中，你很快就会发现可以重复使用之前建好的片段。

如今，当我们着手构建网站的页面时，面对的已经不再是一张白纸，以及各种独特标记和样式的组合。我们能够通过系统已有的原子、分子、有机体和模板创建一个完整的页面。由于整个网站共享相同的代码，如果要调整“发布日期”的行高，那么不必挨个修改每个页面各自的日期数据条目；只需更新一个原子，这个改变就会自动应用到整个系统中。

如今，Pattern Lab 文档网站成为了由构建网站的各个单元有机结合而成的动态系统。它对品牌颜色、字体、商标规格甚至动画都进行了分类。因此，Pattern Lab 成为了最完整而高

效的网站文档记录方式之一。它不仅是一个极好的开发平台，也起到了团队成员之间进行沟通的通用语言的作用。

不要觉得前端开发是在原地绕圈。假如只是构建不同的网页，工作只会随着时间推移变得越来越难。但是如果专注于运用类似 Pattern Lab 的工具构建系统，那么实际上工作会随着时间的推移变得越来越简单。请不要担心这样会让我们失业！一旦从枯燥的网站独立页面开发工作中解放出来，就能有更多的心思来研究如何完善我们的系统。这也正是我要介绍 JSON 模式的原因。第 17 章不会再介绍一般类型的示例，而会以 Red Hat 网站为例来具体说明，JSON 模式是如何将 Red Hat 从类似 Pattern Lab 的系统转变为完全集成的设计系统的。这个系统不仅可以生成样式规范，同时也是内容编辑系统的基础，而且还是网站的主要渲染引擎。

第 17 章

Red Hat文档

回顾为 Red Hat 网站开发新的设计系统的长达一年的过程后，我意识到整个项目是从一个文档需求开始的，并在此基础上发展壮大。管理层最初的需求是为网站最通用的模块开发一个样式文档，并在 Red Hat 的其他站点之间复用。项目开始时只有一些 Sass 文件片段和 Hologram 样式文档，但它所孕育的却远不止这些。

17.1 阶段1：静态的样式文档

就像第 15 章中提到的，Hologram 是一种文档工具，它在代码中寻找有特殊标记的文档注释块，并把这些注释块转换成样式文档。从 Nicole Sullivan 那里了解到 Hologram 以后，我感觉它应该是为正在开发的组件编写文档的完美工具。

Hologram 使用起来很方便快捷。每一个组件和布局都有自己的 Sass 文件片段，因此我们将文档块写在文件的最开头。文档块记录了很多东西，从组件的设计意图到它的功能和局限性，甚至还包括示例的 HTML 代码。示例的 HTML 代码可以帮助我们迅速了解组件的功能，而不用触发 CMS 系统，从而使得原型开发、跨浏览器测试和视觉还原测试更简单。

有了组件库和布局库之后，我们还花了很多时间来写入门文档。该文档不仅包括如何设置样式文档，还包括如何使用、如何贡献新的内容，以及如何发布新的版本。在这方面，Hologram 是个很不错的工具，因为我们可以简单地为文档创建一个文件夹，然后在其中创建 Markdown 文件，Hologram 就会基于页面的类别将这些 Markdown 文件的内容写入样式文档里。

起初一切都进展得很顺利，直到我们突然记起最初的目标是创建一个样式文档，其中说明如何组合代码来创建出各种常用的模板。这并不困难，只需要创建一个模板文件夹，在其中创建 Markdown 文件，将 Markdown 文件连接到模板中使用的组件和布局中即可。Markdown 文件包括了对栅格、对齐、主题值等的描述。创建完成后，就可以将标记复制到 Markdown 文件，用它来说明诸如品牌 logo 如何显示、如何用 HTML 标记实现视频区块，以及特色活动区块中有两个活动和有三个活动时代码的差异。

问题是，每次将 HTML 标记从一个地方复制到另一个地方，标记被改变的概率就会无限增加。我们很快就发现这种方式有多么难以维护，因此很快开始寻找新的解决方案：对 HTML 标记片段使用单一源，并按需重用。不幸的是，Hologram 本身不支持任何模板语言。2014 年 10 月，我在这个项目的 Github 主页上提出了一个问题，“将 html_example 处理为 ruby 的视图模板，从而可以重用模板的继承关系”（https://github.com/trulia/hologram/issues/159），如今仍待解决。

我们提出的解决方案是，在文档文件传递到 Hologram 之前，就使用 Twig 模板引擎对其进行处理。我很惊讶地发现，Twig 处理 Markdown 文件完全没有问题。下面是文档的简单示例：

````
<!-- cta.docs.md -->

---
hologram: true
title: CTA Component
category: Component - CTA
---

- A "Call-to-Action" component contains one or more CTA buttons.

## Primary Button

```html_example
 {% include "cta.twig" with {'type': 'primary'} %}
```

## Secondary Button

```html_example
 {% include "cta.twig" with {'type': 'secondary} %}
```

```html_example
<!-- cta.twig -->
<div class="rh-cta" >
 <a class="rh-cta-link" data-rh-cta-type="{{type}}" href="#">
  CTA Button
</a>
</div>

```
````

从上述单个文档文件中可以看出，我们可以借助 Twig 的 `include` 函数，使用不同的参数，在文件不同的地方输出同样的 CTA 标记。结果文件可以直接传递给 Hologram 进行常规处理：

````
<!-- cta.docs.md -->

---
hologram: true
title: CTA Component
category: Component - CTA
---

- A "Call-to-Action" component contains one or more CTA buttons.

## Primary Button

```html_example
<div class="rh-cta" >
 <a class="rh-cta-link" data-rh-cta-type="primary" href="#">
  CTA Button
</a>
</div>
```

## Secondary Button
```html_example
<div class="rh-cta" >
 <a class="rh-cta-link" data-rh-cta-type="secondary" href="#">
  CTA Button
</a>
</div>

```
````

如果要修改 CTA 标记，只需更新单个 Twig 模板文件，整个系统就可以使用新的标记了。

17.2　阶段2：重写Pattern Lab

上文提到的方法很适合为单个组件写文档。但我们也逐渐需要为布局写文档了，比如卡片布局。卡片布局很简单，你可以将多个组件放在一个有外边界的盒子中，并在其中应用主题和背景。但是，如果使用 Twig 导入卡片布局标记，怎样才能知道卡片中使用的是哪个组件呢？我们可以创建一堆卡片布局，每个卡片布局引用不同的组件，但这不是又在复制标记代码吗？

实际上 Twig 支持对 `include` 语句传递参数，于是有了下列解决方案：

```
{% set template = 'cta.twig' %}

{% include template %}
```

这正是我们需要的能力，使用它可以创建各种卡片布局，每个布局的设置和内容都不一样。

```
<!-- card.docs.md -->

---
hologram: true
title: Card Layout
category: Layout - Card
---

{%
 set data = {
   "theme": "dark",
   "components": [
     {
       "template": "image_embed.twig",
       "src": "my-image.jpg"
     },
     {
       "template": "cta.twig",
       "type": "primary"
     }
   ]
 }
%}

{% include card.twig with data %}

<!-- end card.docs.md -->

<!-- card.twig -->

<div class="rh-card data-rh-theme="{{theme}}">
 {% for component in components %}
   {% include component.template with component only %}
 {% endfor %}
</div>

<!-- end card.twig -->
```

上述代码做了一些稍微复杂点的工作，这里把它分解一下，因为从卡片发展到组、区块甚至更复杂的结构的过程是整个系统构建的基础。

```
{% set

 data = {
   "theme": "dark",
   "components": [
   ...
   ]
 }

%}
```

我们的第一项任务是创建一个描述如何构建系统的数据集。正如所见，我们写了一些包含两个属性的代码，分别是 theme 和 components。theme 只是一个字符串，而 components 是包含两个对象的数组，分别是模板属性和其他一些与模板相关的属性。也许你认为这看起来有点像 JSON，没错。虽然我们一开始使用 Twig 数据对象，然后逐渐转换为以 JSON 数据表示的工作流，但是二者的逻辑是一样的。

下一步需要使用上文设置的数据来导入卡片模板：

```
{% include card.twig with data %}
```

最后是卡片的 Twig 模板文件，看一看它是如何处理我们传入的数据的 :

```
<div class="rh-card data-rh-theme="{{theme}}">

 {% for component in components %}
   {% include component.template with component only %}
 {% endfor %}

</div>
```

首先遇到的是 {{theme}} 变量，这一步很简单。我们从 theme 属性获取了 dark 值，并赋值给 data-rh-theme 属性，过程跟 CTA Twig 文件中的一样。

第二个 Twig 片段稍微复杂一些。{% for component in components %} 表示对数组 components 中的每一个组件（即 component）进行如下操作：引入从每个数组的 template 属性中找到的模板，并将整个数组作为新的数据上下文传递到该模板中。因此在如下的代码中，我们首先引入 image_embed.twig 文件并传入图片地址，然后引入 cta.twig 并设置 type 的属性为 primary。

```
{
  "template": "image_embed.twig",
  "src": "my-image.jpg"
},
{
  "template": "cta.twig",
  "type": "primary"
}
```

如果这些对你来说很熟悉，那也许是因为你刚读过前面的章节，其中介绍了如何使用 Pattern Lab 为布局引入的每一个原子组件和模块传递数据上下文。如果你猜想我们是不是……没错，最终我们创建了自己的 Pattern Lab。

很多同事在意识到我做了什么时会笑我，问我为什么不一开始就使用 Pattern Lab。我当时的回答也很简单：走到这一步是很自然的。我们从一些基本功能出发，随着需求的变更不停地添加新功能，最终和 Pattern Lab 就很像了。不过最后我们发现，还是自己开发的系统更符合我们的需求。

Hologram 不仅提供了存储模式库的地方，还可以快速创建样式文档，并为安装、最佳实践和部署指引等建立文档。我们自己的系统和 Pattern Lab 之间还有一个很大的区别，那就是我们通过数据来决定引入的模板，而不像 Pattern Lab 一样把模板信息硬编码在布局或模块组合生成的有机体中。

这意味着我们可以对任意内容重用布局，而无需为每一个组合创建新的布局。另外还有一个需要注意的重点：一旦渲染流程开始，我们对数据以外的标记是没有控制权的；我们没办法为某个组件的特殊情况改变布局文件的资源顺序，或为另外一个组件添加额外的容器 div。这些限制最终会使我们的系统变得健壮和强大。但在达到这个目标之前，我们需要放下一些重担。

17.3　阶段3：分拆模式库和样式文档

随着模式库和样式文档不断扩大，让我头疼的一个地方是，因为它们使用同一个 Git 仓库，所以我们不断推送对样式文档的小调整和大改动，然而这些改动对模式库没有丝毫影响。我们已经对每次迭代都制订了发布计划，而每次改动后都会推送一个预发布的标签。这造成了不少困扰，因为在用新的预发布标签去麻烦后端开发人员之前，我们需要确认新的代码是否已经对模板、样式或 JavaScript 生效。我们对 Git 仓库做了大量的改动，而最终能够看到的效果实际上只是样式文档的几个页面，这也让人感觉有点混乱。

我们的解决方案是将模式库和样式文档分拆到不同的 Git 仓库中。这样做带来的好处比我们想象的还要多。

最直接的好处就是，我们可以全天候地把对样式文档的修改推送到自己的 Git 仓库，而不用担心影响生产环境下的服务器。现在模式库收到的拉取请求少多了，我们可以有更多精力来关注这些改动如何影响生产环境和样式文档，二者都是通过 Bower 加载模式库的。

第二个好处是，现在我们的生产环境和样式文档对模式库的依赖是平等的。当样式文档中创建了一个新的内容块设计时，我们只能通过 JSON 数组来编辑 HTML 标记。我们没有办法为某个一次性的特殊情况修改标记或添加自定义的 CSS。这意味着，对于样式文档中创建的所有内容，我们都有自信在不修改模板的情况下使其在生产环境中重现。

当前生产环境中使用的是 CSS 和 JavaScript，但它有自己的基于 PHP 的模板渲染引擎来处理较为繁杂的工作。我们能够保证把每个组件和模板都原样复制到 PHP 里面，但是这种方式非常依赖人工操作，而且对模式库的每一次改动都必须复制到生产环境中。

这个问题其实很常见，大部分 Pattern Lab 用户可以使用内置的 Mustache 模板创建各种设计、模板和页面，但还需要将编译后的 HTML 标记交给后端开发工程师来实现后端模板。但这并不代表他们喜欢这种方式！我敢肯定，他们希望只修改模式库里的原子组件，CMS（内容管理系统）就能立即根据改动重新渲染页面。

幸运的是，我们已经开始使用 Twig，Twig 本来就是一种 PHP 模板。而且已经有一些开发者正在努力创建模式库和 CMS 渲染引擎之间的链接 。

17.4 阶段4：创建统一的渲染引擎

Drupal（一个基于 PHP 的内容管理系统）的前端开发者面临的最大麻烦之一是，无法在不依赖 Drupal 自身的情况下重写 Drupal 日益复杂的渲染流程，Drupal 就是通过该流程创建 HTML 标记的。我们要么被迫编写静态的 HTML 标记，在最终执行时却被忽略；要么直接使用 Drupal 创建的 HTML 标记，并在开始写样式前就要先在 CMS 中创建内容。

然而把模式库和样式文档分拆之后，我们发现自己其实创建了一个全新的基于 Twig 的渲染引擎，该引擎可以用于 Drupal 和我们自己的样式文档。向 Twig 渲染函数传递同样的 JSON 数据集和标记即服务（或者设计 API），会一直返回同样的结果。现在，我们的模式库可以安装到任何支持编译 Twig 的系统中了。如果 JSON 数据的收集和格式化是正常的，那么渲染出来的 HTML 也是完全一致的，无论 JSON 数据是通过静态文件还是复杂的 CMS 系统传递的。

当然，我们的挑战在于，要知道正确格式化的数据是什么样的。每个模板接受哪些属性？这些属性的类型是字符串、数字还是布尔值？哪些属性是必需的？有没有属性对接受的参数值有限制？如何确保传递给 Twig 函数的参数是有效的，并且能够生成正确的 HTML ？对这些问题的尝试解答将我们引至最伟大的发现之一：JSON 模式。

JSON模式

简而言之，JSON 模式描述了一个数据集。具体而言，在我们的模式库中，JSON 模式描述了正确地渲染一个模板所需要的数据。它列出模板的所有变量、哪些变量是必需的、变量的数据类型，以及对变量的值的所有限制。

如果我们的卡片布局有一个 `theme` 变量，那么对应的 JSON 模式会提示我们 `theme` 变量是必需的，而且取值只能是 `dark` 或 `light` 这两个字符串之一。如果我们试图忽略 `theme`，或者给模板递字符串 `gray`，那么卡片布局就会返回验证错误提示。

我们可以使用 JSON 模式提示卡片布局可以包括一系列不同的组件，而且还可以通过它来验证用于渲染某个页面区域的复杂 JSON 数组，所有的验证只需要运行一次。下面的代码可能包含一个有效的卡片布局，但是其中的 CTA 组件却是无效的，因为其中 `type` 的值不能是 `tertiary`：

```
{
  "template": "band.twig",
  "theme": "dark",
  "components": [
```

```
        {
          "template": "cta.twig",
          "type": "tertiary"
        }
      ]
    }
```

JSON 模式允许我们通过引用来描述可以放到卡片布局、组布局和特定内容块里面的 CTA 组件，因此无需复制 JSON 模式文件。而且和 Twig 一样，我们只使用单一的规范的数据源。因此，通过给不同的组件（小到按钮、大到布局）创建 JSON 模式，我们得到了非常复杂且全面的 JSON 模式文件，通过它就能够验证所有想要渲染的内容块。

有了这些 JSON 模式作为武装，任何模板所需的数据，开发者一看便知，而且他们也可以在数据发送到共享 Twig 模板引擎之前，验证自己收集到的数据的有效性。这是对旧版本系统的一个重大提升，因为再也不用小心翼翼地修改输出的 HTML 标记，直到它完全符合样式文档了。他们可以自信地说，只要数据是有效的，渲染引擎就会输出正确的 HTML 标记。不幸的是，这种方案还有一个问题。

开发者仍然肩负重担，他们要在 CMS 中为页面里的标题、图片和文本段落创建几十个所需的输入框。就算所有的输入框都创建完成，他们还要在渲染之前吃力地将这些标题、图片和文本段落映射到 JSON 数组。他们也无法重用这些输入框去收集每个模板文件的数据。每当需要收集文本、链接和 CTA 按钮所需的数据时，他们都要在 CMS 系统中创建更多的输入框，而且相应地也需要更多的数据库条目。

但我们还是很幸运的，因为 JSON 模式还有几招没有使用。

17.5　阶段5：自动创建新模式

虽然我不是一名经验丰富的 PHP 后端开发人员，但我和他们一样痛恨重复的工作。我可以真切地感受到这些开发人员的痛苦，仅仅为了匹配写好的模式，他们就要重复地配置几十个输入框。我不禁思考，如果我们能够在完全不需要后端介入的情况下，就把在 JSON 模式、模板和样式上所做的工作程序化引入到 CMS 系统中，那我们的系统该有多强大！

如果我们不需要等待下一个为期三周的迭代再开发资源来实现新的模板，如果我们能够从根源上避免“翻译损失”的问题（即开发者需要先对模式功能作假设，之后需要一个迭代来解决出现的一大堆问题），那岂不是很好吗？

Jeremy Dorn 的 JSON 编辑器（http://jeremydorn.com/json-editor/）可以根据 JSON 模式的属性生成 HTML 表单，并在更新表单域时返回 JSON 数据。我们在模式库中使用这个工具来操作 JSON 模式、模板和样式，通过实时视觉反馈来观察属性改变对渲染结果的影响。由于 JSON 编辑器有将 JSON 模式转变成表单的功能，而且在这方面我们取得了很大的成功，

我们相信这个方案同样也可以成功地用于任何使用我们设计系统的平台，包括 Drupal 和 WordPress。

当开发人员有机会为 Drupal 创建自动化的 JSON 模式引入流程时，我们对 JSON 模式能力的信心就得到了回报。借助于 JSON 模式，短短几周内，我们有了一个将组件和布局转换为 Drupal 实体的自动化系统。现在，我们就可以为 CTA 组件创建单独的 Drupal 实体了，并在每次包含组件时引用 Drupal 实体及其对应的数据库字段。

创建模式

我们也想把 Red Hat 品牌下已经创建的很多其他模式利用起来，它们都是一些把不同布局和组件组合在一起的方法。一个模式指定了使用的栅格布局、组件和某些属性的值。如果需要创建一个品牌图标墙，包括五张为一行的内嵌图片，我们可以简单地使用一个模式而不用手动设置。

事实证明，页面模式也可以用 JSON 模式来描述。每一个内容块模式都描述了这个模式可以接受的数据，这看起来与内容块的 JSON 模式很相似。但是它比 JSON 模式更严格，并且不一定会提供所有的背景、布局和组件的所有可选项。

借助于简单的 JSON 模式数据集的力量，我们不仅能够设计组件的原型，还可以定义用于渲染原型的模板，甚至定义模板所需的数据模型，以及 CMS 中用于搜集数据的 UI！

我们当初决定采用 JSON 模式，它的功能强大，但也有各种限制，而这一切让我们建立起如今的系统。这个系统开始时仅仅是一些设计元素和一部分希望用于实现的 HTML 标记，最终变成了一个从原型设计到页面渲染的高效流水线，并省去了很大一部分重复的工作。它使我们的前端工作能够完整地定义 CMS 中的整个流程，包括数据收集、数据存储、页面渲染和样式内容等。

第 18 章

总结

在这个项目的整个过程中，我所学到的关于前端架构的最重要的知识之一就是，对于任何优秀的架构而言，只要未到最后一刻，你的工作就不会结束。这个项目从非常简单的结构开始，创建出了一个满足当时需求的架构。但是随着时间推移，我们的需求发生了变化，项目也随之变得更复杂，而一个成功的案例会给其他领域提供有用的参考。

有时候回顾整个过程，想想花了那么长时间才达到目前的状态，不由得心里会问：“为什么不在一开始的时候就这么做呢？为什么我们要经历那么多迭代，而不在第一天就把最终产品的架构定义好呢？难道这说明我们失败了，或者我们的工作毫无价值？”不是这样的，这一点我非常肯定。

随着对前端架构的理解越来越深入，我可以肯定的是，从项目开始到现在所达到的高度，所需的时间会越来越短，而且所经历的迭代也会越来越少（虽然迭代是流程中必不可少的一部分）。作为前端架构师，我们的职责是认清目前的优势和劣势，并预测可能出现的机遇和问题。经历过后方能预测，曾经低估才能理解。我们所能展示的最大能力就是对前端开发过程的深刻理解。

因此，如果这是你第一次制定架构计划，请记住你将要迭代很多次！不要把你所有的希望寄托于一个单独的解决方案、框架或者平台，除非它一次又一次地被证明是有效的。如果你从事这个工作很长时间了，那么你已经知道，我们仍然需要不断地迭代和尝试新的事物。如今我们不会再见到使用各种表格布局和静态设计稿搭建出来的新网站，因此永远不要以为你在过去的几个项目中用过的工具仍旧是完成当前工作的最佳选择。

但最重要的是，如果你也想要成为一名前端架构师，请记住你永远不是在孤军作战。业界有很多人投身于前端架构的艺术，不管他们的头衔如何。

不要害怕寻求别人的帮助，不要害怕分享你的知识，也不要害怕站在台上鼓励他人从事这个领域。最后，不管你做什么，在任何情况下都不要害怕把它们全部写到书上。

作者介绍

Micah Godbolt，前端架构师，作家，播客播主，世界级开源大会的培训师和演讲师。他在个人博客（https://micahgodbolt.com）中经常大力推广前端架构、Sass、视觉还原测试和基于模式的设计方法。他出生于太平洋西北地区，目前和妻子以及两个孩子定居于波特兰的郊区。

Micah 曾经受聘于 Red Hat（红帽）公司。但是，本书中所表达的观点仅代表作者本人，并不代表作者就职过的任何公司，包括 Red Hat。

封面介绍

本书封面上的动物是粉头虫莺（学名 Cardellina vericolor）。它是一种小型雀形目鸟，生活在高海拔地区（1800~3500 米），主要分布于危地马拉和墨西哥南部。

成年粉头虫莺平均长约 12.7 厘米，重约 10 克。它的鸣唱声接近“啾啾”和“吡吡”的金属声。虽然只有雄性会鸣唱，但两性都会通过一种更低的啁啾声来与配偶保持联系。

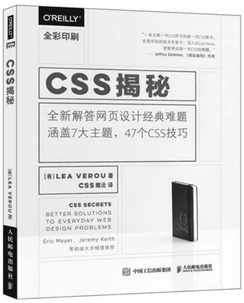
除非共同喂养小鸟，否则很少能看到两只粉头虫莺待在一起。它们喜欢栖息于温带雨林里的低矮茂密的灌木丛中，但是这种类型的栖息地在它们的活动区域里正逐渐消失。可以预见，随着危地马拉森林面积大幅减小，它们将会跟很多其他物种一样，陆续迁徙到萨尔瓦多。

雄鸟通常在二月开始鸣唱，并且会持续几个月。与此同时，雌鸟开始在地面用松针和苔藓筑巢。交配之后，雌鸟通常会产下 2~4 枚鸟蛋，并孵化约 16 天。随后雏鸟会由父母哺育10~12 天，之后雏鸟就要离开鸟巢，学会独自捕食昆虫。

O’Reilly 封面上的许多动物都已濒临灭绝，但它们的存在对世界至关重要。想要了解如何帮助它们，可以登录 animals.oreilly.com。

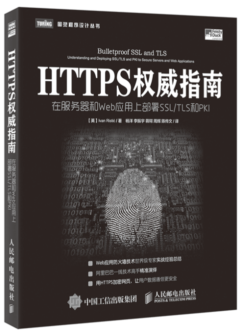
本书封面图片以 Wood 的 *Illustrated Natural History* 一书中的黑白版画为基础。

延 展 阅 读



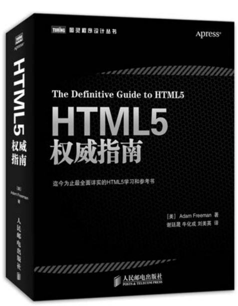
- CSS一姐Lea Verou作品，全新解答网页设计经典难题
- 《CSS权威指南》作者Eric Meyer倾情作序

书号：978-7-115-41694-0
定价：99.00 元



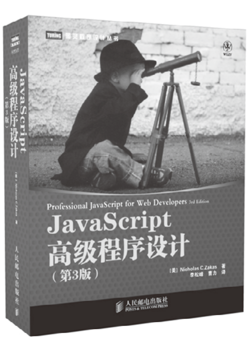
- Web应用防火墙技术世界级专家实战经验总结，阿里巴巴一线技术高手精准演绎
- 用HTTPS加密网页，让用户数据通信更安全

书号：978-7-115-43272-8
定价：99.00 元



- 精彩呈现500多个实战代码示例及主流浏览器实现效果图
- 贴心汇聚HTML5和CSS3中所有属性、元素和函数的简明参考表

书号：978-7-115-33836-5
定价：129.00 元



- 一幅浓墨重彩的语言画卷，一部推陈出新的技术名著
- 全能前端人员必读之经典，全面知识更新必备之佳作

书号：978-7-115-27579-0
定价：99.00 元

O'REILLY®

前端架构设计

前端架构是一系列工具和流程的集合，旨在提升前端代码质量，并实现高效、可持续的工作流。对于大型Web项目，前端架构师和软件架构师同样不可或缺。

本书作者通过Red Hat公司真实案例分析以及以往经验积累的实用技巧，系统总结了前端架构的四个核心，详细展示了新的前端开发准则，将Web开发提升到了一个新高度。

前端架构四个核心：

- **代码——如何实现系统架构中的HTML、CSS和JavaScript**
- **流程——构建高效并且防止出错的工作流所需要的工具和流程**
- **测试——为网站搭建稳固基础**
- **文档——规划好系统设计蓝图**

前端架构师职责：

- **体系设计——清晰描绘产品和代码的最终形态**
- **工作规划——制定完整开发工作流**
- **监督跟进——保证项目高效率完成**

“前端架构是对更高效、更专业的Web开发方法的一种全新理解，Micah重新定义了新时代网站建设过程中我们所能扮演的角色。”

——Christopher Schmitt
Environments for Humans会议主席

Micah Godbolt，前端架构师，作家，播客播主，世界级开源大会的培训师和演讲师。他在个人博客（https://micahgodbolt.com）中经常大力推广前端架构、Sass、视觉还原测试和基于模式的设计方法。他出生于太平洋西北地区，目前和妻子以及两个孩子定居于波特兰的郊区。

WEB DEVELOPMENT/DESIGN

封面设计：Randy Comer　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机／Web开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-45236-8

ISBN 978-7-115-45236-8
定价：49.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱: ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks